# Exploiting inference in temporal planning problems with concurrency

Talukdar, Atif Hossain

*Awarding institution:*
King's College London

# EXPLOITING INFERENCE IN TEMPORAL PLANNING PROBLEMS WITH CONCURRENCY

A thesis submitted to King's College London
for the degree of Doctor of Philosophy
in the Faculty of Natural and Mathematical Sciences

December 2019

Atif Hossain Talukdar
Department of Informatics

# Abstract

Temporal planning has become an increasingly popular area of research in recent years. Much of this is because the modelling of many real world problems requires the consideration of time and how actions interact and depend on one another. A key phenomenon that modelling with time allows us to accommodate when planning is the notion of concurrency, where actions occur simultaneously. Required Concurrency is where certain actions must occur in parallel due to one or more dependencies between actions. Dealing with concurrency in temporal planning requires careful reasoning when making choices about which actions to use during search. Inference can be used to assist in problem solving when there is required concurrency because certain actions *must* occur at the same time. The research presented in this thesis explores how problems of required concurrency can be solved using more inference and less search. Specifically, the work focuses on pair-wise cases of required concurrency. Novel techniques are presented for detecting required concurrency. In addition, new infer and search algorithms capable of exploiting inference from these detections and reducing search are developed. The algorithms for pattern detection, the combined infer and search strategies, along with the inference engine that powers them are implemented as an extension of POPF. We present an evaluation of the practical benefit from this new planner that we call POPI. Additionally, we present a formal analysis provably showing which unique types of required concurrency exist between pairs of durative actions.

# Acknowledgements

I would like to extend my sincere thanks and gratitude to Maria Fox and Derek Long for their supervision, guidance and support. I have learnt a lot from them and their time, dedication and understanding has been essential to the completion of this work.

I would also like to thank everyone in the Department of Informatics, the wider college and beyond, who have supported me during my studies. Finally, I would like to thank my parents, family and friends who have encouraged and helped me to keep on going.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

As a part of Artificial Intelligence (AI), research in Automated Planning has been a growing area of interest, both in terms of theoretical work and the development of planning systems for practical application. Research in temporal planning has become highly motivated; real world problems can be modelled more accurately when considering the durations of actions. The complex and challenging feature to deal with in temporal planning is concurrency where multiple tasks occur at the same time. Required Concurrency is where two or more actions must occur during the same time period in all solutions to a problem (Cushing et al. [2007a]). Where there is required concurrency, existing temporal planners mostly still use search to add an action that must be in any valid solution, given the existence of some other action in the plan. This results in unnecessary search and sometimes exploring paths in the search space that cannot yield a solution. In this thesis we investigate problem domains containing durative actions defined in PDDL 2.1 (Fox and Long [2003]), specifically focussing on situations in which those actions must occur concurrently, at least in part. We then examine patterns that arise when there are pairs of durative actions that execute concurrently in a plan. We provide an analysis of these pair-wise required concurrency situations and propose a pattern recognition based method for detecting required concurrency between pairs of actions in the domain, prior to the planning phase. Some of the patterns of required concurrency presented in this thesis are inspired by those presented in (Cushing et al. [2007a]). A system to use this detection is proposed and described as a driving force for performing temporal inference during planning to reduce search and unnecessary exploration of the state space. We will implement a new planner that can identify these patterns and do the inferences that are possible from the planner being informed of the temporal structure of the patterns and which pairs of actions they apply to.

## 1.1 Research Goals

In this thesis we set out the following four research goals:

1. To identify all of the patterns of action pairs that have required concurrency between them and show what information can be deduced from each one. This specifically refers to all of the pair-wise patterns of actions which are meaningfully different in terms of the combinations of concurrent orderings possible within the structure of each pattern. In Chapter 3, diagrams are presented showing at least one of the configurations in which each distinct pattern can arise.

2. To prove that the catalogue of patterns presented is complete for pair-wise cases, according to the unique set of concurrent orderings that represent each pattern.

3. To define and measure the information content and resulting strength of each pattern in terms of the extra information that can be inferred by a planner from the pattern structure.

4. To develop and demonstrate practical techniques for detecting each of the patterns of required concurrency in domains containing durative actions and also developing algorithms for exploiting the inferences during search in a planning system.

## 1.2 Contributions

In this thesis we make the following contributions.

- A theoretical analysis of pair-wise cases of required concurrency, showing how knowledge of the pattern structure can enable a planner to perform inference. We also show how chains of patterns can enable larger and more powerful inferences to take place.

- Theorems stating a set of pattern sequence combinations that are complete and exhaustive. We show each of the pair-wise patterns presented, correspond to each of these generalised sets of pattern sequences.

- A method for determining the strength of each pattern according to how it is triggered for inference.

- A theoretical method for compiling a pair of actions in a pattern into a single action.

- Using the principles of information theory to propose a method for calculating the gain in bits of information that a planner acquires using knowledge of the patterns structures. This method is used to calculate the information gain for each trigger case of each pattern.

- A method for doing pattern detection for each of the 15 patterns types handled.
- Development of two modified versions of EHC capable of using the inferences of each pattern type when detected.
- Extending POPF to use the pattern detection and inferences methods developed, producing the planner POPI.
- Results and evaluation using domains that include the actions in the patterns identified, showing that there is a sub-class of problems where there are results with fewer state evaluations and better significantly better scaling, with more problems solved.

Some of the material presented in this thesis has been presented in the ICAPS Doctoral Consortium 2016 (Talukdar [2016]) and PlanSIG 2016 (Talukdar et al. [2017]).

## 1.3 Organisation of thesis

The remaining chapters of this thesis are arranged in the following manner. Chapter 2 provides background on relevant literature that we use as the basis for the work in this thesis. The chapter starts with an introduction into Automated Planning and provides some key definitions. We provide details of temporal planning, focussing on required concurrency and describe the modelling of domains using durative actions in PDDL 2.1. We then move onto giving detailed descriptions of temporal planners that are capable of dealing with problems of required concurrency, we use the most relevant of these planners in our experiments for the comparative testing and evaluation of our planning system POPI, detailed in Chapter 7.

In Chapter 3 we move onto the theoretical basis for our work, defining the scope of our work and the assumptions we make. We then present and describe the pattern structures that we handle and illustrate the relationship between the pair of actions for each pattern and the temporal constraints that exist. We explain how a planner knowing the structure of these pattern types gives it the ability to do temporal inference, exactly what those inferences are, and at what time they can occur during plan construction. We also describe how pairs of actions in individual patterns can be linked together in a chain, allowing much larger inferences to be made in certain situations. We illustrate and explain with examples the way in which such chains can exist.

In Chapter 4 we introduce a formal basis for showing that our approach covers a complete set of pair-wise patterns. We provide theorems to prove that the sets of patterns we handle are all of the uniquely different combinations of sequences. We explain how these sets of sequences correspond to the patterns presented in Chapter 3. We provide a method for determining the strength of each pattern trigger case in terms of inference power and show how

the inference power of each trigger case differs the others. Finally, we present a method for showing how pairs of actions of particular pattern types may be compiled into single actions. We provide the criteria for when it is safe to compile a patterns of actions and when it is not and include examples of both.

In Chapter 5, we use an perform an information theoretic analysis of our patterns described in Chapters 3 and 4 and propose a method for calculating the information gained by using POPI's approach for each pattern case, compared to using other strategies. We explain our method for performing this analysis and the other strategies that we compare our strategy against. We show the number of states explored by each strategy and the path in the search tree that is navigated for each of the patterns.

In Chapter 6 we provide detailed descriptions of the methods developed to extend POPF into POPI. We describe the domain analysis used to collect the data needed to formulate the candidates potential pattern detections. We explain how the various parts of pattern detection process occur and how detected patterns are created in and stored in custom data structures. We explain how POPI infers new actions and then present in detail the two novel algorithms, that are modified versions of EHC which have been developed to execute our inferences. We show how these inferences are interleaved with search during the planning phase and present the algorithms for both of these strategies. The first modified verison of EHC is an aggressive strategy for pursuing inference referred to as EHC-AI and the second version is a passive strategy referred to as EHC-PI. We explain what happens in each scenario for both strategies. POPI using either the EHC-AI or EHC-PI strategy results in two variants of the planner that we refer to as POPI-AI and POPI-PI respectively. We conclude this chapter by showing how POPI maintains the properties of completeness and soundness to the same extent that POPF does.

In Chapter 7 we present an empirical analysis of our work, using POPI which implements our techniques. We report on experimental results using domains that incorporate each of the individual pattern types that we handle, and also domains containing chains of patterns. We use problems of increasing difficultly to see how well our planner scales, and compare the performance with other temporal planners described in Chapter 2. We also compare the planners on temporal benchmark domains to also assess competitiveness in problems which do not contain required concurrency.

Finally, in Chapter 8 we conclude the thesis by providing a summary of the work performed and also suggest directions for future research.

# Chapter 2

# Background

This chapter provides a review of planning, concentrating on temporal propositional planning as this is the area of context for the work in this thesis. We provide a description of the Search, Infer and Relax framework proposed in (Hooker [2005]), and how planning may be viewed within this framework. We discuss some planners that use inference to varying degrees since extending the use of inference in temporal planning is the main research focus for this thesis. We also describe other relevant temporal planners that we use for comparing our approach in the empirical analysis in Chapter 7.

## 2.1 Planning

Automated Planning is a model based approach to automate problem solving in Artificial Intelligence. Typically, research is focussed on domain-independent planning, where a planner can solve a problem regardless of the context being modelled in the domain. A planning problem consists of a *domain*, *problem* and a *plan*. Let us start by discussing the basics of what happens in the planning process followed by a discussion of the extensions that have made planning increasingly more expressive and better suited for real-world applications.

The *domain* defines a set of action schemas, which describe the preconditions and effects of each action and their object types. Predicates are statements that describe the relationship between objects of a specified type and are instantiated as propositions with a boolean value. These are the statements which are used to state what the preconditions and effects of each action are. Typically in planning the *closed world* assumption is employed, meaning that propositions that are not assigned a value are treated as being false.

A *problem* describes an initial state and a goal state in the context of some domain using grounded propositional literals. The act of planning is finding a valid sequence of actions that takes us from the initial state to the goal state specified in the problem. A *plan* is a sequence of actions that takes the planner from the initial state to a state containing the goal for some problem instance. A *state* is the description of the world at one point in the planning process,

described using a set of propositional statements. Many planners use a state-based approach to record change and solve the planning problem. In such state-based planners, the actions correspond to a series of state transitions that transform the initial state to a goal state.

Classical Planning involves a fully observable and deterministic state space, where the exact state of the world is always known after applying every action. Each action has one set of preconditions over a set of propositional variables that must be satisfied when it is applied and whose effects are instantaneous and subsequently updated in the successor state. Actions whose effects are immediate are known as *instantaneous actions*.

A classical planning problem only uses instantaneous actions which contain no temporal information or understanding about the duration of actions. Temporal Planning adds a significant aspect to the planning problem, whereby actions are assigned a duration and have start, overall and end preconditions, as well as start and end effects. This means that where the preconditions of one action is achieved by another, there can be a need to interleave the application of the starts and ends of different actions. This creates additional constraints that must be enforced in order to solve the problem.

### 2.1.1 Planning Concepts and Terms

In this section we provide some definitions of the key terms used in classical planning and then explain how they are modified and extended for the temporal case.

**Definition 2.1** (Action)**.** An action is an instance of an operator defined in the domain. An action $a$ is a tuple of the form $\langle operator(\mathrm{a}),\ pre(\mathrm{a}),\ \mathit{eff}(\mathrm{a})\rangle$. $operator(\mathrm{a})$ describes which operator the action is an instance of, $pre(\mathrm{a})$ specifies the preconditions that must be satisfied for the action to execute and $\mathit{eff}(\mathrm{a})$ tells the planner how to update the state after that action has finished and which propositions are now true and which are now false as a result of applying the action.

**Definition 2.2** (State)**.** A state $s$ is an allocation of values to a set of propositional variables $F$. The propositions that are true and false are stated as such and propositional statements not asserted are deemed to be false.

**Definition 2.3** (State transition)**.** A state transition is an operation that changes a state $s$ into $s'$. Typically an action $a$ takes the role of this operation and transition is achieved by using the effects of $a$ to update the values of the state variables in $s'$, on the condition that the preconditions of $a$ are satisfied in $s$.

**Definition 2.4** (State Space)**.** The state space of a problem is the set of all the states that can reached for a given planning problem. The state space can be viewed as a graph, where the nodes are states that have directed edges between them that represent the actions transforming one state into another.

**Definition 2.5** (Planning Problem)**.** A planning problem $P$ is made of an initial state, a set of propositions, some goal states, and a set of actions. The goal states are all the states that contain the goal conditions.

**Definition 2.6** (Plan)**.** A plan is a sequence of actions $\langle a_1, ...., a_n \rangle$ which are applied to generate a series of state transitions that transform the initial state into the goal state. A path between the initial and goal states in the state space is a plan.

## 2.2 Temporal Planning

In temporal planning, the key component added to the problem is duration information. Each action description is extended with a duration specified in its action schema in the domain. The duration of each action is given either as a fixed number of time units or described as set of duration inequalities, where minimum and/or maximum duration bounds are specified. In the latter case, the planner can decide the duration of the action as long as the duration constraints are respected. These temporally descriptive actions are called *durative actions* (Fox and Long [2003]) and are described in the next section.

### 2.2.1 Durative Actions

A durative action includes the description of start and end preconditions as well as invariant conditions. The start and end preconditions must be satisfied at the start and end of the action respectively and the invariant conditions must hold for the duration of the action. The start and end effects are also specified separately and describe the change in the propositional and numeric variables that take place at the start and end of the action respectively. Since durative actions can describe change in numeric variables at the start and end and the action duration is known, it is possible to model linear continuous change for each numeric variable. The increase or decrease can be calculated at each time point during an action's execution, by taking the end time of the action, subtracting its start time and then dividing this value by the action duration. This gives the amount of change in the value of a numeric variable per time unit, and the value of the numeric variables can be calculated for all the time points in the action's time interval.

### 2.2.2 Snap Actions

In order to model start preconditions and effects along with end preconditions and effects, some temporal planners split each action into two corresponding start and end actions. These are referred to as *snap* actions. The preconditions for the start actions represent the facts needed to be true to start the action. The start effects represent those effects occurring instantly after the start snap action has been applied. The same principle applies for the end

snap actions. The end preconditions must be true for the action end to be applied; its effects are also updated in the state variables of the resulting state. Snap actions describe discrete change to state variables and are also referred to as *happenings*. Invariant constraints are specified as *over all* conditions that need to remain true in all intermediary states generated between the application of the start and end of an action. Any external action that negates any of these over all conditions cannot be used in the time interval between the start and end of the action with the invariant constraints.

## 2.3 Concurrency

### 2.3.1 Required Concurrency

Temporal planning problems can be classified as one of two categories, those that are temporally simple and those which are temporally expressive, as described by Cushing et al. [2007a]. The latter of the two being problems that have *required concurrency*.

Required concurrency is where two or more actions must occur simultaneously at least in part. This is due to causal links between the starts and ends of actions that form precedence constraints between the endpoints of different actions. It is this type of required concurrency that this thesis investigates. Where one of these actions is added to the plan, the required concurrency indicates a type of inference is possible. This is because each action in the pair can only occur together with the other, therefore the second action's inclusion can be inferred after adding the first action. The work of this thesis shows how such inference can be made in these types of situations and can be exploited in a useful way.

An example of required concurrency is in the `matchCellar` domain presented by Coles et al. [2009c], where light is generated by lighting a match and is only available for a finite period of time, in which a fuse can be repaired; this domain is in turn derived from (Long and Fox [2003]). In this domain there is required concurrency between the `mend_fuse` and `light_match` actions. The goal of the problem is to fix a fuse in a cellar. Light is needed to fix the fuse, and the only way to see in the dark and mend the fuse is to light a match. The start of the `light_match` action causes the 'light' fact to be true and is deleted at its end. The `mend_fuse` action has 'light' as an invariant condition and therefore it must start after `light_match` and end before it, for the two actions to occur successfully and satisfy the goal. The required concurrency between these two actions is illustrated in Figure 2.1.

### 2.3.2 Temporal Coordination

It is possible for two actions to execute simultaneously in all the solutions to a problem, without there being causal links between the actions involved. This can be in cases where there are other causes for the temporal interaction between actions to become more complex,

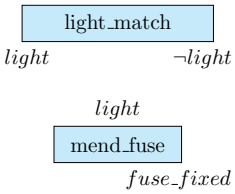



Figure 2.1: Required Concurrency in matchCellar domain (Coles et al. [2009c]).

due to temporal coordination being needed, for example in the case of short action durations. One instance of this situation is where other preceding actions must be coordinated carefully, such that the timing of the concurrent actions can actually occur and complete successfully. For example, in the `marsRover` domain shown in Appendix A, a rover must work with a drone to perform an inspection of rock samples at a particular location, $w3$. The rover requires light to be shone on $w3$ by the drone so it can do its inspection and achieve the goal. There is required concurrency between the drone's `shine_light` and the rover's `inspect` actions. The `inspect` action must occur entirely within the *envelope* of `shine_light`. An envelope is a time limited window of opportunity created by an event or action. In this example, the opportunity is the resource 'light' being made available by the `shine_light` action for the duration of its execution. The rover is at $w1$ and the drone at $w2$ in the initial state. The rover must use the `navigate` action to go from $w1$ to $w3$ and the drone must use the `fly` action to go from $w2$ to $w3$. Let us assume that all locations are of equal distance to each other and that the drone flies faster than the drone drives between locations. In order for both the drone and rover to arrive at $w3$ at times close enough together to solve the problem, the drone should be coordinated to leave after the rover starts its journey but before it arrives. Temporal coordination is needed because the drone can only hover over $w3$ for a limited amount of time due to battery consumption, so it needs the rover to be there soon after it arrives. There needs to be enough flight time left for the drone to continue hovering over and inspect $w3$ when the rover shines the light on $w3$.

### 2.3.3 Optional Concurrency

One type of optional concurrency is where there is a solution to a problem containing a pair of actions that must occur concurrently, however there is also an alternative sequential plan to reach the goal as well. This means that the planner can take either the path with the forced concurrency between actions or the sequential path to solve the problem depending on how it works. Another type of optional concurrency is where two or more actions which could appear sequentially in time are scheduled to occur in the same time interval, at least in part. In this case, it would have to be that the two actions were independent in achieving their preconditions without either one deleting the other's preconditions before it is to be applied.

A planner may use this type of optional concurrency to shorten the plan *makespan*, which is the overall duration of the plan.

## 2.4 Modelling in PDDL

In planning, the language that has become the standard for modelling domains and problems across the field is the *Planning Domain Definition Language* (PDDL), originally introduced by (McDermott et al. [1998]). This version of PDDL which was used in the first International Planning Competition (IPC) (McDermott [2000]), allows modelling of classical planning problems where instantaneous actions are used and all actions in a plan take place sequentially. The action in Figure 2.2 shows an instantaneous action defined in PDDL. The action specifies the operator name, the boolean variables and their types for the predicates contained in the preconditions and effects of the action. The preconditions of the action must be satisfied before the action can be applied and all effects are updated in the state variables of the subsequent successor state. Actions defined in this format are often referred to as STRIPS (Fikes and Nilsson [1971]) actions.

```
(:action pickup
:parameters (?t1 - tool)
:precondition (and (handempty) (ontable ?t1) )
:effect (and (not (handempty)) (not (ontable ?t1)) (holding ?t1))
)
```

Figure 2.2: Pickup action for tools on a table.

As previously discussed durative actions describe how actions can be embedded in time. An example of a durative action named `Act_A` from the `patternsD` domain presented in Appendix B is shown in Figure 2.3. For temporal propositional planning, a durative action differs from an instantaneous action in that preconditions are renamed as conditions in order to be able to split them into *at start*, *over all* and *at end* conditions. The effects are also split into *at start* and *at end* effects. The purpose of these changes is to allow for durative actions to be able to specify at what point during an action's execution in time, the different conditions need to be true and when effects take place. Additionally, a crucial component added to durative action definitions is its duration. These temporal actions can be defined with fixed durations, as shown in Figure 2.3 using the `=` symbol. Alternatively, durative actions can have flexible durations described using a set of constraints, which define the lower and upper bounds for the duration. These duration inequalities are described using the `<=` and `>=` symbols.

In the example shown in Figure 2.3 we see that the action is defined with a fixed duration of three time units. This action takes two parameters, `?a` and `?b`, both of `typeA`. Two objects

of `typeA` from the problem instance are used to ground this action for use during planning. We see that the action has three start preconditions which are `(active)`, `(next ?a ?b)` and `(ready ?a)`. These start preconditions must be true in order for `Act_A` to become applicable and before it can start its execution. There are two end preconditions which again includes `(ready ?a)` and also includes `(q ?a)`. The end preconditions must both be true before the time at which the action ends. In the action effects we see that `(active)` fact is deleted as a start effect, as well as `(p ?a)` and `(ready ?b)` being made true. These effects are instantaneous and take place immediately when the start of the action is applied. The same applies for the end effect which makes `(active)` true again; this fact becomes true instantaneously when the end of `Act_A` is applied.

```
(:durative-action Act_A
 :parameters(?a ?b - typeA)
 :duration(= ?duration 3)
 :condition(and (at start (active)) (at end (q ?a)) (at start(next ?a ?b))
  (at start (ready ?a)) (at end(ready ?a)))
 :effect(and (at start(p ?a)) (at start(not(active))) (at start(ready ?b))
         (at end (active)))
 )
```

Figure 2.3: Example of durative action defined in PDDL 2.1.

The set of PDDL languages has grown considerably since its use in the first IPC, with various extensions to allow modelling of problems using different features. Some of these include PDDL 2.2 (Edelkamp and Hoffmann [2004]) used in the 4th IPC (Hoffmann and Edelkamp [2005]), PDDL+ (Fox and Long [2006]), and PDDL 3.0 (Gerevini et al. [2009]).

## 2.5 Plan Construction

### 2.5.1 Total Ordering

Total ordering exists where the execution order of actions in a plan, is the same as the order in which the actions were added to the plan during plan construction. For temporal planners using total ordering, the timestamp of each new action appended to the plan during its construction, is later than the timestamps of all the previous actions added to the plan up until that point. COLIN (Coles et al. [2012]) is an example of a temporal planner which uses total ordering, as does its predecessor, CRIKEY3 (Coles et al. [2008]).

### 2.5.2 Partial Ordering

Partial ordering is based on the idea of least commitment, where the planner will not commit to the ordering of actions in its plan until it becomes necessary. POPF (Coles et al. [2010])

is a planner which uses this technique and implements a partial ordering mechanism. POPF orders actions such that an operator which fulfils the precondition of another is ordered first. This results in the causal links between actions being satisfied. In cases where an action has an effect on the value of a numeric variable or is dependant on one, the planner must enforce the ordering of this action and the time it occurs. This is an example of where action ordering must enforced to ensure plan consistency and that constraints are not violated. The partial ordering mechanism of POPF therefore allows reordering of actions between which there is no causal link. However, the planner still requires that some action orderings be fixed in the plan, to preserve the specified constraints. For the implementation of its partial order machinery, POPF adds actions to the plan in the order it selected them, and reorders the execution order as necessary. The execution order of actions is changed by altering their start times.

## 2.6 Search-Infer-Relax Paradigm

The Search-Infer-Relax (S-I-R) framework (Hooker [2005]) shows how solutions to combinatorial optimisation problems, including planning problems, can be solved using different combinations of search, inference and relaxation. Planning often uses a combination of these techniques as part of the various strategies it employs for solving these problems. In fact, a typical combination of these techniques in a planning strategy often involves some level of each of these three components. *Search* is the process of iteratively applying applicable actions to produce potential successor states. *Relaxation* is the process of solving an easier version of the problem to help choose actions for state generation, helping to avoid the need to perform a complete search of the state space and hence allowing larger problem instances to be solved. *Inference* is a logical deduction that adds a new fact about the world, given some previous statement. In state-based planning, inference is constantly being used to update the values of the state variables using the effects of the action used to reach each state. Inference provides with certainty, information about what must happen when actions and events occur and assists the search process by informing the solver about what options there are for it to explore next, given the new state of the world.

We can see an example of how search, inference and relaxation are used in combination to solve a problem in a state-based planning approach. A common approach is that a planner uses some strategy, like Enforced-Hill Climbing (described in Section 2.6.1), to iteratively apply actions and build successor states. Relaxation is used to solve an easier version of the problem to help decide which actions to use, to lead the planner closer to the goal. Finally, inference is used at the point where the action chosen is applied, by using its effects to update the affected state variables in the successor state.

### 2.6.1 Search

Given a combinatorial optimisation problem, search is the process of iterating through each possible operation from a state to produce the next reachable state. Each operation adds constraints to the problem, in order to reach a version of the problem that is potentially easier to solve (Hooker [2005]). For search to occur in a state space represented as a tree, as is the case for planning, all of the different options (eg. actions) that can be explored from one state in order to reach the next state need to be represented.

There are many search algorithms that have been developed to allow exploration of a search space. In planning, search is often guided using heuristics to allow an informed and goal directed search process to occur. However, as well as an informed search, there is also uninformed search, where the state space is expanded through brute force application of all possible actions with no guidance until a state satisfying the goal is reached. This type of search is really just an ordered expansion of the state space, but has the benefit of being *complete*. A search strategy is complete if it can find a plan to solve the problem, whenever a solution to the problem exists. Depending on the state representation and implementation of the solver, breadth-first search can expand a vast number of states, since there is no heuristic used to choose which action to apply next. Two of the most well known algorithms that do this are breadth-first and depth-first search. Breadth-first search uses all applicable actions to generate all the successors at the next level of the search tree, then progresses to the first of those successors and does the same again for that state and all other states on that level from left to right. The same is then repeated on all subsequent levels of the search tree, one at a time until the goal is reached. Depth-first search is similar, but the difference being that from each state, the tree is expanded as far down as possible before coming back up to build the other successor states. Breadth-first builds across the states space first, then moves down, whereas a depth-first approach moves all the way down before moving across the search tree. Depth-first search is not usually complete in its search, unless it has a mechanism for termination on branches for repeated states. Breadth-first search is complete on problems with a bounded width; in planning this corresponds to branching on action choices for a finite set of grounded actions.

Heuristic based search algorithms build the search spaces using guidance provided by the heuristic about which action is best to apply at each state in order to get closer to the goal. This way the search space can be explored more efficiently and concisely, allowing a more intelligent and strategic approach that can be utilised in larger problems, where searching entire state spaces is simply not possible with limited time and memory resources. There are various search algorithms that make use of heuristics; the following section explains Enforced-Hill Climbing.

#### Enforced-Hill Climbing

Enforced-Hill Climbing (EHC) (Hoffmann and Nebel [2001]) is a search strategy which we modify for the work in thesis, and we now describe how this strategy currently works. EHC is a greedy version of the Hill-Climbing strategy, where at a state $s$, an action is applied to generate a successor state $s'$. The state $s'$ is evaluated to estimate its distance to the goal. Only if the heuristic value (distance to the goal) of $s'$ is strictly lower than $s$, is $s'$ progressed to. If this is the case, then the other successor states from $s$ are not constructed, even if there is a better heuristic successor. The search queue for $s$ is cleared and state $s'$ becomes the new current state and the same process continues until a goal state is reached. If at any point during this process no successor is generated that has a better heuristic value than the current state, the planner temporarily resorts to a local search called best-first search to find a lower heuristic state. Best-first search is a variation of Breadth-first search. Algorithm 1 presents the Enforced-Hill Climbing strategy as it is described by Coles and Smith [2007]. The pseudo code for Algorithm 1 is reproduced from (Coles and Smith [2007]) and is included here for convenience.

**Algorithm 1:** Enforced-Hill Climbing

```
1  open_list = [initial_state];
2  best_heuristic = heuristic value of initial_state;
3  while open_list not empty do
4      current_state = pop state from head of open_list;
5      successors = the list of states visible from current_state;
6      while successors is not empty do
7          next_state = remove a state from successors;
8          h = heuristic value of next_state;
9          if next_state is a goal state then
10             return next_state;
11         if h better than best_heuristic then
12             clear successors;
13             clear open_list;
14             best_heuristic = h;
15         place next_state at back of open_list;
```

### 2.6.2 Inference

Inference is the process of taking a set of propositions and using defined rules, to be able to add with certainty, further propositions to that set. If we have a proposition $p$ and the following inference rule: $p \rightarrow q$, then if $p$ becomes true, we infer that $q$ is also true. This inference that $q$ is inferred from $p$ may be written in the form $p \vdash q$. This inference follows from an inference rule known as *modus ponens*. Constraint Programming uses inference as a mechanism to solve problems, as does planning. Although typically it seems like planners mainly use search along with heuristics to solve problems, in fact planners constantly are performing inference,

the most basic of which is during state progression. State-based planning systems apply an action to generate the next state. The act of updating the values of state variables during state generation using the effects of the action used to transition to the that state is inference. Another form of inference is always applying the start snap actions before their respective ends and always applying the ends of actions that have started, even if the goal has been satisfied. These tasks may seem trivial and obvious, however they represent important logical rules that must always be followed. Given a state $s$ and an action $a$, this can be written as an inference rule $s(a) \rightarrow s'$, where $s'$ is the state generated by applying $a$ at state $s$. Inference is a powerful mechanism utilised in various bodies of work including that of Coles et al. [2009b], which proposes the use of more inference in a forward search framework. Planning approaches utilise inference in various ways. The work by Allen in planning as temporal reasoning (Allen [1991]) describes how planning systems can be converted into an inference process.

### 2.6.3 Relaxation

Relaxation involves solving a simplified version of a problem in order to estimate the distance to the goal in the actual search space. A *heuristic* is an estimated distance of how far a state is from a goal state, in terms of the number of actions that need to be applied to reach the goal. In state-based planning, the addition of an action to the plan is attained by performing a state transition. Heuristics are developed from solving the relaxed problem at a state, and are used to inform the decision making process of the planning system as to what action choice is best to make next and likely to take it closer to a goal state. In planning, the *delete* relaxation is commonly used to produce an easier version of the problem being solved (Hoffmann and Nebel [2001]), as a means of estimating the distance to reaching the solution of the original problem. The Heuristic Search Planner (HSP) (Bonet and Geffner [1999]) is an example of a planning system which uses this relaxation. The problem is relaxed by ignoring all of the delete effects of the actions. The *relaxed plan* is the sequence of actions in the solution to the relaxed version of the problem. The heuristic distance of a state $s$ to the goal $g$ in the actual search space, is the number of actions in the relaxed plan constructed at $s$. This heuristic estimate is useful because if a problem $m$ is relaxed into a problem $m'$ and there is no solution found when solving $m'$, then we immediately know that the original and harder problem $m$ cannot be solved. In addition, if there is a solution to $m'$, then this can provide direction on which search option to use to solve $m$. A heuristic is described as being *admissible* if it never overestimates the distance/cost to reaching the goal of the problem.

There has been a lot of work conducted in the development of heuristics in recent years. Relaxations used to construct heuristics in planning include the Relaxed Planning Graph (RPG) (Hoffmann and Nebel [2001]) for classical planning problems where domains are purely propositional. Planners that use the RPG for heuristic guidance, such as the Fast Forward

(FF) (Hoffmann and Nebel [2001]) planner, construct an RPG at each state in the search space. The RPG is made up of alternating fact and action layers which record the facts which are true and the actions that are applicable at each stage. When the relaxed version of the problem is being solved, the delete effects of actions are ignored meaning that when a fact is made true, it remains true. This also means that when an action in the RPG becomes applicable in an action layer because its preconditions are satisfied, it remains applicable in all future action layers. Ignoring delete effects in this manner means the problem is easier and quicker to solve than the original problem. The length of the relaxed plan in terms of the number of actions it contains, is used as the estimated distance from the actual state, where the RPG was built, to the goal. The Metric Relaxed Planning Graph (MRPG) (Hoffmann [2003]) is extended from the RPG for problems with numeric state variables, but behaves the same as the RPG for propostional variables when ignoring delete effects.

The Temporal Relaxed Planning Graph (TRPG) (Coles et al. [2008]), is also an extension of the RPG and behaves like the RPG for ignoring propositional delete effects. However, the TRPG introduces the use of timestamps in the layers of the graph, and records the earliest times that actions become applicable and facts become true. This is because durative actions are used in the TRPG, where each action has a duration assigned to it. Planning with durative actions means that they are each divided into start and end snap actions. The length of the relaxed plan is again used as the estimated distance to the goal. Another relaxation used in temporal planning is an extended TRPG for temporal problems that require the use of envelope actions in the plan (Coles and Coles [2017]). Examples of other heuristics developed based on relaxation in planning, include the Causal Graph (CG) heuristic (Helmert [2004]) which uses hierarchical problem decomposition, and a cost based RPG (Sroka and Long [2014]) for metric sensitivity.

## 2.7 Simple Temporal Networks

Planning systems that are capable of solving temporal planning problems need to find a sequence of actions to solve the problem as well as a time schedule that makes them temporally viable, according to the temporal constraints of the problem. Some state-based temporal planners use a temporal network of some kind to record the temporal constraints that have accumulated from the actions applied so far up to the current state. In order to check that a state is temporally consistent, a Simple Temporal Network (STN) (Dechter et al. [1991]) can be used to record the temporal constraints between time points. A copy of the updated STN is maintained at each state with the constraints added for the most recent action used to transition the planner into the current state. The constraints that exist are of the form $x < y$, where $x$ and $y$ are two snap actions and there exists a precedence constraint of $x$ before $y$.

The STNs contain duration information and record the minimum and maximum durations for each action, which are equal in the case of fixed duration actions. The STN is used to check that the state is temporally consistent after each state generation. If there exists a negative cycle between any pair of actions where the minimum duration is greater than the maximum duration, then the state is temporally inconsistent and is pruned from the search space. Negative cycles in an STN can be found in polynomial time, allowing for efficient detection of a temporally inconsistent state. The polynomial time complexity of this task is what allows STNs to be integrated into the search process. STNs are used by many planners including CRIKEY (Halsey et al. [2004]), CRIKEY3 (Coles et al. [2008]), COLIN (Coles et al. [2012]) and POPF (Coles et al. [2010]).

## 2.8 Temporal Planners

### 2.8.1 Temporal GraphPlan

The Temporal Graphplan (TGP) (Smith and Weld [1999]) algorithm is a temporal extension to GraphPlan (Blum and Furst [1995]). GraphPlan is a partial order planner capable of constructing shortest distance plans. TGP uses an extended version of the STRIPS action definition, by first letting each action have a start time and a duration assigned to it. All of an action's preconditions must be satisfied at the beginning before its application; there is no individual separation of start and end preconditions. These preconditions must remain satisfied for the action's entire duration unless it modifies them and all effects take place only at the end. The temporal actions TPG utilise prevent actions occurring concurrently if one of them deletes any of the preconditions or effects of the other. This method essentially locks the resources it needs and produces for its duration, not allowing interfering actions to happen at the same time. This type of modelling pre-dates the emergence of durative actions defined in PDDL 2.1 (Fox and Long [2003]). This extension of TGP allows it to do a similar mutual exclusion reasoning that GraphPlan could do but with actions embedded in time and assigned with differing durations. This allows TGP to solve temporally expressive problems. Other temporal planners that effectively convert durative actions into an extended STRIPS action similarly to TPG include SGPlan (Chen et al. [2006]), SAPA (Do and Kambhampati [2001]) and LPG-td (Gerevini et al. [2006]).

Although TGP can deal with concurrency, it imposes restrictions; the planner is limited in that it treats predicates whose values it changes at the end of an action, as having an unassigned value and is neither truth or false. This means other actions that may want to use that propositional fact as a precondition cannot while the first action is happening, meaning no concurrency between these two actions can happen. An example of this is the `matchCellar` domain problem shown in Figure 2.1, where lighting a match in a dark cellar provides light

for a limited time, in which the `mend_fuse` action must take place is order to achieve the goal of fixing the fuse. The `matchCellar` domain problem is an instance of required concurrency that we refer to as pattern type A for the work in this thesis. This pattern is illustrated later in Chapter 3.

### 2.8.2 CPT and eCPT

Constraint Programming (CP) can be used to solve combinatorial search optimisation problems by representing the problem domain as a Constraint Satisfaction Problem (CSP) and using a method of constraint propagation to prune the search space so that it can be efficiently searched. CPlan (van Beek and Chen [1999]) is a constraint programming planner that uses this approach. Constraint Programming is an area of research that has made substantial use of inference. Other more recent CP based planners are CPT (Vidal and Geffner [2004]) and eCPT (Vidal and Geffner [2005]). CPT translates a PDDL planning problem into a CSP, where the problem is represented as a set of constraints over a set of variables. These planners work by first setting a bound $k$ on the length of the plan and assuming from the initial state that the plan cannot have more than $k$ steps. The planner attempts to solve the problem using a maximum of $k$ actions and if it cannot, must backtrack and can increase $k$. This bound is needed because the planners convert the problems in constraint programs which have a fixed number of variables. CPT uses canonical plans, where each grounded action is used no more than once. eCPT is an extension of CPT that lifts this restriction such that grounded actions can occur more than once. eCPT implements extra domain-independent inference machinery and is capable of solving simple problems with pure inference and no search and in most cases backtrack free. These include problems from the benchmarks which are: Blocks, Depots, Driverlog, Ferry, Gripper, Logistics, Miconic, Rovers, Satellite and Zeno domains as mentioned in (Vidal and Geffner [2005]). eCPT finds optimal plans by using an iteratively deepening search, meaning that it does repeated search using different bounds. The $k$ bound is increased if a plan with the existing bound cannot be found, meaning that when a plan is found it corresponds to the minimum number of actions possible for the planner to find a solution. Both CPT and eCPT do inference through consistency checking, making sure that all constraints are satisfied while solving the problem. eCPT is also a planner that is capable of solving problems with required concurrency.

The model on which CP-based planners base the representation of the action structures for the construction of the plan, is using sequences of discrete time intervals. In CPT and eCPT, its constraint model is based on these time intervals. The planner calculates the largest common time interval that can be used to divide all of the actions durations in the problem. This is the Greatest Common Divisor (GCD) of all of the actions durations, which are all fixed durations, since CPT and eCPT do not handle duration inequalities. The GCD is then used

to establish the length of each time interval in which the plan is constructed. Each action is assigned the number of time intervals needed to match the length of its duration, which is why all actions durations in the problem must be divisible by the interval unit length. For example, if a problem has 3 actions with durations of 2, 4 and 6, then the GCD for the time intervals is 2, since this is the largest integer to divide all 3 action duration values. We note that this representation of time is used by CPT and eCPT, however alternative representations are possible in other constraint-based planners. A key difference of CPT and eCPT compared to state-based planners, is that for example in POPF (described in Section 2.8.11), a state is an assignment of values to variables at a particular time (each state has a timestamp assigned to it). The work of Verfaillie et al. [2010b], reviews the Constraint Network on Timelines (CNT) framework (Verfaillie et al. [2010a]) and illustrates its usefulness and how it supports modelling of problems in planning, scheduling and constraint programming based approaches.

### 2.8.3 Temporal Fast Downward

Fast Downward (FD) (Helmert [2006]) is a planner that does heuristic search for solving classical planning problems. The Temporal Fast Downward (TFD) (Eyerich et al. [2009]) planner is an extension of this system for solving temporal planning problems. TFD is a planner that uses total ordering during plan construction and is limited to handling fixed duration actions. This means that TFD cannot handle duration inequalities, which is where some actions in the domain have flexible durations, the values for which are decided by the planner. For its temporal structure, TFD is similar to how SAPA operates. TFD works by exploring a search space where it either applies an action if its start preconditions are satisfied, or it advances time to the next end action. When TFD applies a start action, it is constrained to apply the action at $epsilon(\epsilon)$[1] time after the current time. This stops it from delaying the application of a start action to a later time point. This means that if there are two actions $A$ and $B$ that must occur together, if B must start at a time point relative to the start time of A and the gap is more than epsilon time, then TFD cannot solve this problem. TFD maintains a priority queue for the ends of actions which have been started but not yet completed. These action ends are placed in time order, given the duration of the actions. This means that when TFD chooses to apply an end action, it must be the end action that is earliest in the priority queue. Furthermore, TFD ignores the preconditions of end actions, and as such cannot correctly solve required concurrency problems, where end preconditions are part of the temporal structure creating the required concurrency relationship.

In order to understand exactly what kind of required concurrency TFD is capable of handling, let us consider the following example. Figure 2.4 shows two actions $A$ and $B$.

[1]Epsilon time is is defined as being the smallest time gap possible, that is used to separate two actions, where one action makes a fact true and the other action requires it as a precondition.

Action $A$ is 10 time units in duration and $B$ is 5 time units. $A$ provides resource $p$ at its start and deletes is at the end. $B$ requires $p$ as an invariant condition and so $p$ must persist for the duration of $B$'s execution. With the goal of achieving $g$, actions $A$ and $B$ are needed. TFD is able to solve this problem by first applying the start of $A$ and then the start of $B$, which can only occur epsilon time after $A$ start. Given that the start of $A$ takes place at time *0*, the end of $A$ in the priority queue has a timestamp of *10*. The start of $B$ is at time $\epsilon$, and so the timestamp of the end of $B$ is time $5 + \epsilon$. Following the application of both the starts of $A$ and $B$, there are no other actions to start for TFD, so it can only advance time to the end action with the earliest timestamp in the priority queue, which is the end of B. This followed with the application of the end of $A$ which is the next and only remaining action in the priority queue.

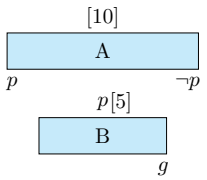



Figure 2.4: Action pair with required concurrency relationship that TFD can handle.

For the type of required concurrency seen in Figure 2.5, to achieve the goal $g$, TFD uses the same approach. Action $B$ start is applied at time *0* and $A$ start at time $\epsilon$, this means that in the priority queue action $B$ end will have a timestamp of *10* and $A$ end a timestamp of $5 + \epsilon$. This means that $A$ end must be first from the actions in the priority queue, since its timestamp is earlier than that of $B$ end. Since $B$ end provides $q$ which is the precondition for the end of $A$, TFD is not able to apply the ends of actions $A$ and $B$ in the correct order. Figure 2.5 is example of a type of required concurrency problem that cannot be solved by TFD.

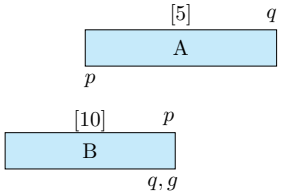



Figure 2.5: Action pair with required concurrency relationship that TFD cannot correctly handle.

### 2.8.4 TPSHE

TPSHE (Jiménez et al. [2015]) is a temporal planner that compiles a temporal problem fully into a classical planning respresentation. TPSHE can solve problems of required concurrency, but is specifically limited to dealing with *single hard envelopes* (SHE) (Coles et al. [2009c]) problems. A single hard envelope exists when an action makes a resource available for the duration of its execution, by making true a fact at its start and deleting it at the end. This action is referred to as an envelope action. If another action needs this resource as a precondition, it can occur concurrently with the envelope action, and is referred to as a content action. Where there is a problem of this type, TPSHE can construct a plan provided that one envelope action, has the effects to satisfy every invariant condition of the encapsulated content action. This means that the content action cannot have a set of invariants satisfied by different envelope actions. In TPSHE, the definition of a single hard envelope is modified to not only include an action that achieves an effect at its start and deletes it at its end, but also a second action that needs that effect as an invariant. This means in TPSHE a single hard envelope exists if both the envelope and content actions exist and not just the envelope action. The content action must have a shorter duration than the envelope in order for both actions to be able to occur concurrently. This type of required concurrency corresponds to the pattern type A which will be presented in Chapter 3 of this thesis. Figure 2.1 shown earlier in Section 2.3.2 illustrates an example of a single hard envelope from the `matchCellar` domain. TPSHE does not handle other types of required concurrency and all durative actions in the domain must have fixed durations, as the planner does not handle duration inequalities. TPSHE has been implemented via alterations to the Fast Downward planner.

### 2.8.5 TP(K)

TP(K) (Jiménez et al. [2015]) implements an adjusted version of the TEMPO algorithm (Cushing et al. [2007a]) that partly compiles temporal planning problems into classical ones. The planner compiles each durative action into two classical actions, one that represents its start component and one for its end component. The TEMPO algorithm which TP(K) modifies is based on the idea of *lifted temporal states*. Each of these lifted states record the values of variables, a set of currently executing actions that have started but have not finished, a time variable and the temporal constraints on the time variables. The time variable records the time of the most recent action which is either a start or end action. During the planning phase TEMPO progresses search by using two rules which the authors refer to as *Fattening* and *Advancing Time*. Fattening is for the application of a start action, which is added to the set of currently executing actions. This can only occur if the start preconditions and invariant conditions of the action are true in the current state and its start effects do not delete the invariants of other executing actions. Advancing time is for ending an action, at which point the action is removed from the set of currently executing actions. The end time of the action

is calculated by adding the duration of the action to its start time.

TP(K) uses the two classical actions compiled from each durative action, to practically implement the Fattening and Advancing Time rules in TEMPO, for the application of action starts and ends. The TP(K) planner is implemented as a modified version of Fast Downward. In Fast Downward each node of the state space records the state along with what is referred to as bookkeeping information, however Fast Downward is a classical planner. For this reason TP(K)'s modifications include altering the node representation, so that its bookkeeping information also records currently executing actions, and temporal constraints through the addition of an STN. The $K$ value of the planner is a parameter which is set at run time and passed to the planner along with the domain and problem files. This value is the bound given to specify the maximum number of currently executing actions there can be. The $K$ bound means that in practice this implemented version of the planner is not complete.

### 2.8.6 STP(K)

STP(K) (Furelos-Blanco et al. [2018]) is a temporal planner that also compiles temporal planning problems into classical ones. The planner builds a plan using the classical representation of the problem and at the same time ensures that an associated set of temporal constraints are maintained. STP(K) is an extension of TP(K) to deal with required concurrency problems that have simultaneous events. The key feature introduced by STP(K) is its ability to produce plans for temporal problems that must have simultaneous events. This means that the planner can schedule durative actions such that more than one effect from different actions can occur at the same time. In order to achieve this, STP(K) splits every event into three parts. The first part is for action ends to occur, the second part is where simultaneous effects occur and the last part is where actions are started. These three parts are referred to by the authors as the *End Phase*, *Event Phase* and *Start Phase* respectively. The end phase takes place just before an event happens and is the part where currently executing actions end. The event phase is the part where the simultaneous occurrence of the actual event happens. In order to make sure the simultaneous event is valid, the preconditions of the event are checked and effects of the event are also applied at this stage. The start phase occurs just after the event phase and is the point where the planner verifies that the invariants hold for the currently executing actions, which have started in this phase.

Like the TP(K) planner, STP(K) is also implemented using modifications to Fast Downward. There are various features that STP(K) and TP(K) both use including the addition of an STN to the bookkeeping information of the search nodes, in order to record temporal constraints. STP(K) also records the set of currently executing actions at each node. The $K$ value is again a parameter that is passed to the planner at run time and is the maximum

number of currently executing actions there can be. This means that if there is a problem that requires more than $K$ actions to occur concurrently, STP will not be able to find the solution and hence is also a planner that is not complete. Both the TP(K) and STP(K) planners only handle problems with fixed durations and do not handle duration dependent effects.

### 2.8.7 TFLAP

TFLAP (Sapena et al. [2018]) is a temporal forward search planner that uses partial ordering. It is based on another forward partial order planner called FLAP2 (Sapena et al. [2016]). The idea of partial ordering actions in forward search planning was used earlier in POPF (described in Section 2.8.11), which also use this idea. TFLAP is built such that its parser is capable of handling all features of PDDL 3.1 (Helmert [2008]), however the actual planning part of the system can only handle PDDL 2.1 along with TILs (Time Initial Literals), which is a feature defined in PDDL 2.2. The planner uses A* search (Hart et al. [1968]) to navigate the state space. TFLAP makes use of classical planning heuristics to evaluate each state, and does not utlise information about action durations to decide which node to visit next. As a result, a limitation that TFLAP suffers from in domains that contain dead-ends, is that it can cause the planner to enter *plateaus*. A plateau is entered when the planner reaches a state in the search space, where all of the successor states appear to be no better than the current state, according to the heuristic. As a partial order planner TFLAP is able to avoid ordering actions unless it becomes necessary due precedence constraints between actions. An example of this is where one action achieves the precondition of another. TFLAP is also a planner able to deal with problems of required concurrency.

### 2.8.8 ITSAT

The Implicit Time SAT (ITSAT) (Rankooh et al. [2012], Rankooh [2013]) planner is a system developed to solve temporal planning problems by translating them into SAT encoded problems. This SAT representation of the problem uses variables and clauses in its encoding. Among these, one variable is used to represent the start of an action and a second variable to represent its end. A third variable is used to record an action which is currently executing. A standard SAT solving system is used to attain what the authors refer to as an *abstract* plan from the SAT representation of the problem. This version of the plan is abstract since it does not include the action durations. This abstract plan is in turn relaxed, to which the actions durations are then inserted. The temporal validity of this plan is checked against the corresponding STN. If the STN is found to contain a negative cycle, then this information is added to the SAT representation of the problem. The updated SAT encoding of the problem can then be used by the SAT solver to solve the problem again using the same approach. The idea is that the information about the negative cycle from the STN of the last plan produced, can be used by the solver to stop it from outputting plans with STNs, that have negative

cycles which are the same or similar. ITSAT iteratively applies this approach until it finds a temporally consistent plan. The ITSAT planner has also been modified to incorporate other encodings described in (Rankooh and Ghassem-Sani [2013]).

### 2.8.9 CRIKEY

CRIKEY (Halsey et al. [2004]) is a planner that decouples the planning component of the problem from temporal scheduling task when solving a temporal planning problem. A modified version of Metric-FF (Hoffmann [2003]) is used to solve the planning problem and the STN is used to schedule those actions to make a temporally valid plan for the original problem. Where possible, CRIKEY solves temporal problems in this fashion, avoiding in many cases the need to solve the entire temporal problem as a single task. Where there are problems with temporal envelopes, where one action must occur concurrently within the duration of another, as is the case in the `matchCellar` domain, is not possible to decouple the planning and scheduling components of the problem in the same way. The altered approach is described further by (Halsey et al. [2004]). CRIKEY has also been extended into CRIKEY3 (Coles et al. [2008]) which is a planner with the ability to handle all language features of PDDL 2.1, level 3 (Fox and Long [2003]) and is able to solve temporal propositional and numeric problems. CRIKEY3 is the predecessor of COLIN (Coles et al. [2012]) (described in Section 2.8.10). Another successor to CRIKEY is CRIKEY$_{SHE}$ (Coles et al. [2009c]) which is capable of detecting actions that are single hard envelopes. The `matchCellar` domain is again an example of where a single hard envelope exists, as shown in Figure 2.1. In this example, the `light_match` action is the envelope action and `mend_fuse` is the content action.

### 2.8.10 COLIN

COLIN (Coles et al. [2009a, 2012]) is a temporal numeric planner and a successor to CRIKEY3. COLIN is capable of dealing with continuous linear processes, designed to solve planning problems with numbers embedded in time. In particular, COLIN is able to reason with continuous numeric effects and linear change. The planner uses linear programming (LP) to measure the rate of change for numeric variables between states, given the amount of time passed. It also uses LP to check that temporal constraints are satisfied when numeric change has occured. Furthermore, the planner uses an STN to check that duration constraints are satisfied at each state. COLIN is a total order planner, which means every action added to the plan is appended to the tail of the plan constructed so far. If the planner needs to change the order of any actions in the plan, it must backtrack out of the states in which those actions have been applied. The actions must then be applied again in the desired order. This applies even when changing the order of actions between which there is no precedence constraint.

### 2.8.11 POPF

POPF (Coles et al. [2010]) is a forward search partial order planner capable of solving temporal planning problems according to the full language specifications of PDDL 2.1 (Fox and Long [2003]). POPF is the successor to COLIN and can therefore also solve temporal numeric planning problems but the difference is that COLIN employs a total ordering during plan construction and suffers from the problems of early commitment to action ordering choices. POPF employs the principles of least commitment and only orders actions in the plan when it is necessary due to some temporal precedence constraint. Actions added in a particular order during plan construction that do not interact with one another can be rescheduled to be in a different order for the final plan. This is on the condition that all the temporal constraints are respected. POPF is a temporal planner and like COLIN it solves problems with durative actions. The planner divides each durative action into two snap actions, one for its start and the second for its end component as does COLIN and CRIKEY. This means that the number of states generated is doubled compared with if snap actions were not used. This is done in order to allow concurrent problems and those requiring temporal coordination to be modelled and solved.

The planner uses a partial order (described in Section 2.5.2), to record precedence constraints, and also uses an STN at each state to record duration constraints of actions and to check the temporal constistency of the state. If a state is found to be temporally inconsistent, then the plan constructed so far at that state (partial plan) is invalid. A state with an invalid partial plan is detected if there is a negative cycle in the STN and these states are pruned from the search space. If the problem contains numeric change, then an LP is used to check plan validity as is done in COLIN. The primary search strategy used by POPF is Enforced-Hill Climbing (described in Section 2.6.1). If EHC fails to find a solution, the planner uses best-first search as its secondary strategy, in the case of POPF this is actually A* search. POPF uses the TRPG as its heuristic function to guide the search process. A *helpful action* is an action in the first action layer of the TRPG, which either achieves a goal fact or a precondition of an action achieving a goal fact. POPF extracts the list of helpful actions from the TRPG constructed at each state in the search space. The list of helpful actions are ordered arbitrarily, and the first one in the list is used to generate the successor state.

POPF can perform a more efficient search using *compression-safety* (Coles et al. [2009b]). An action is compression-safe if the following criteria are satisfied: the action has no end preconditions that are not a subset of the invariants, it has no negative end effects and no start effects that depend on the duration of the action. When a durative action is compression-safe, POPF can apply the start action as a state transition producing a successor state and only schedule the end of the action as required, to satisfy the action's duration constraints and

any other temporal constraints. The key point is that only one state is generated explicitly in the search space for a compression-safe action.

In contrast to CPT, in POPF grounded actions can be applied multiple times in a plan. However, generally this is not common in practice in the temporal propositional case, which is the scope of this thesis. This is because after a grounded action's effects are updated on the propositional state variables that it modifies the first time, applying the same grounded action again would be redundant as it causes no additional change to occur on those state variables. This is unless a different action is applied in between the two applications of the first action, where the second action changes the value of one or more of the propositional state variables updated by the first action.

In order to ensure that the invariants of an action are true following the application of the action's start component, POPF uses *regression of invariants*, meaning that all invariant conditions of an action are regressed and treated as start preconditions as well as being invariant conditions. The reason for this is that the planner makes sure that the invariant of an action, which must hold true for its duration, have been made true by some other action. The only exception to this rule is when an invariant of an action is also a start effect of that same action. For example if an action $A$ makes a fact $p$ true, and $p$ is also an invariant of $A$, then this is a special case where $p$ does not need to be regressed, since action $A$ itself makes it true before it needs it as a condition. The process described is for regressing invariants in the temproal propositional case. This process becomes more complex when regressing invariant conditions that must hold over numeric variables, however we do not describe this here as this thesis does not investigate numeric planning.

Following the development of POPF, there have been various extensions to POPF, each of which have capitalised on POPF's comprehensive set of functionalities. Some recent extensions include OPTIC (Benton et al. [2012]) for temporal planning with preferences, POPF-TIF (Piacentini et al. [2015]) for planning with timed initial numeric fluents and POPCORN (Savas et al. [2016]) for planning with control parameters.

### 2.8.12 Other Temporal Planning Systems

The planning algorithm TEMPO (Cushing et al. [2007a]) describes a planning system for problems of required concurrency. An adaptation of this algorithm was implemented in the TP(K) planner described in Section 2.8.5. The development of TEMPO was soon followed by an evaluation of temporal domains (Cushing et al. [2007b]) in order to show that the temporal benchmarks at the time were all without required concurrency and hence could all be solved with sequential solutions. UPMurphi (Della Penna et al. [2009]) is another temporal planner

which uses A* search (Hart et al. [1968]) and is capable of dealing with continuous processes. Other temporal systems that exist are the YAHSP set of planners which include YAHSP2 (Vidal [2011]), DAE-YAHSP (Khouadjia et al. [2012]) and YAHSP3 and YAHSP3-MT [Vidal, 2014].

## 2.9 Summary

In this chapter we have reviewed the basics of Planning, specifically focussing on temporal planning. We have discussed the notion of required concurrency and problems of temporal coordination. We have described the Search-Infer Relax framework, its components, which includes the role that inference plays, and how planning can be viewed within the context of this framework. We have reviewed some key planners, including eCPT which uses a constraint programming based approach for doing inference, and also POPF, which this thesis builds on. In Chapter 3, we discuss the types of pattern structures that are handled in this thesis and the inferences that they enable.

# Chapter 3

# Patterns and Temporal Inference

We now move on to explaining the first part of the theory for the research in this thesis. We start by discussing the types of required concurrency that we propose dealing with and explaining what can be inferred from each type and when. Each type of required concurrency is referred to interchangeably as a *pattern* or *pattern type*, which we define as being a pair of durative actions in the domain, whose schemas describe either a one way or two way set of dependencies between the actions. These causal links can be recorded using simple temporal networks constraints. This definition is subject to the restrictions described in Section 3.1. A *pattern instance* always refers to the lifted non-grounded pair of operators in the domain detected as having a required concurrency relationship between them. No grounded actions are recorded as pattern instances, as these are detected in the domain before action grounding. In addition, we define a *trigger action* to be a grounded instance of an operator from a pattern instance that is added to the plan and is detected as being part of a pattern instance; during search this detection allows the planner to do inference.

As mentioned in Chapter 1, the goals of the research include finding actions in pattern structures, which when triggered during search enable inference to occur, reducing the amount of search needed. We describe each of the pattern types and the inferences they enable. Each pattern type along with its associated STN are explained and depicted diagrammatically, showing what actions and constraints must be found via search and which ones can be inferred. This will provide us with a better understanding of what each pattern means relative to one another, along with why and how the temporal orderings enforced by each pattern type's structure are different, given that they all exist within the context of pair-wise required concurrency. It also further motivates and shows us why pair-wise cases are a very interesting set of scenarios to deal with and that it does indeed warrant deeper investigation.

## 3.1 Scope

We focus on pair-wise cases of required concurrency, however we acknowledge that this feature can exist amongst larger groups of actions. The reason for concentrating on pair-wise cases is because we are interested in understanding the minimal cases in which required concurrency can occur in single instances, in its different variations and how this can be exploited for reducing search and to what extent. The aim is to be able to exploit cases of required concurrency with minimal cost, while attempting to maximise the inferential gain. We will see later that for some pattern types, there is some choice about how actions may be applied in accordance with how the pattern is triggered for inference. We observe that there is a trade-off between the amount of computational resources put into performing pattern detection, with only the potential that there will be some benefit from inference during planning. Therefore, when considering more than two actions in a pattern of required concurrency, it stands to reason that the initial outlay in computational expense generally increases for its pattern detection. Even if patterns with three or more operators are detected in the domain, there is no increased chance that they will be triggered for inference during planning, compared with the smaller pair-wise patterns. This strengthens our motivation for restricting ourselves to the two action pattern types. Additionally, there are numerous scenarios that require concurrency between two actions. In temporal planning, well known examples include the 'matchCellar' domain, where a 'mend_fuse' action must occur concurrently with a 'light_match' action in order to fix the fuse. Furthermore, although we restrict to pair-wise cases, this is only for single instances, we will see later how chains of linked pair-wise pattern instances can lead to arbitrarily large collections of actions with required concurrency.

The work of this thesis is in temporal planning, hence we use PDDL 2.1 which is a standard version of PDDL for modelling temporal planning problems and for expressing our patterns of actions with required concurrency. It is important that we first set out the scope of our work and the conditions under which we detect patterns of required concurrency. We limit the modelling of patterns to using a subset of the features provided in PDDL 2.1. This is because given the expressivity of the language, the types of required concurrency that we handle are already intricate in the combinations of predicates needed to construct their pattern structures as we will see in section 3.2. Restricting the features of PDDL used according to the assumptions below, allows us to state with certainty what patterns are possible, while still being able to construct a comprehensive set of patterns. Furthermore, the pattern types that we will present will be implemented in the planner that we develop in this thesis, POPI, which will be explained in detail in Chapter 6. POPI is developed as an extension of POPF and subject to the same restrictions in the languages features that it handles. POPF does not handle negative or disjunctive preconditions, or conditional effects, therefore neither does POPI.

We make the following assumptions:

1. All instances of each pattern type contain exactly two durative actions according to the language specifications of PDDL 2.1 (Fox and Long [2003]).

2. The actions on which we do our pattern detection do not have negative or disjunctive preconditions.

3. The actions that we do pattern detection on do not have conditional effects. In addition to not being handled by POPF, required concurrency cannot always be detected with certainty at the domain analysis stage using our approach. This is because if a predicate that is part of the pattern structure is a conditional effect of one of the actions in a potential pattern, we cannot guarantee that there will be required concurrency during planning, if the action with the conditional effect is applied.

4. All predicates that make up the pattern structure of each pattern type shown in Section 3.2.2 must be achieved by in the format of the diagrams shown, by only the two actions in the pattern. If this is not the case, then it is discarded as a pattern in order to prevent detections in the domain which have concurrency but might not be cases of required concurrency.

5. Each action in an instance of any pattern type must be a grounding of a different operator, unless it is an instance of a reflexive pattern type, in which case both actions must be different instances of the same operator.

6. During planning, if a pattern is triggered, the inference can only take place if the planner determines that there is only one applicable grounded instance of the inferred operator from the pattern, that can be applied. If not, then there is a choice between two or more actions, meaning inference cannot occur.

7. Each set of actions in a single pattern instance, or set of pattern instances chained together, are either all applied or none at all.

## 3.2 Pattern Structures and Temporal Constraints

This section discusses the patterns of required concurrency to be dealt with, that will be identified by recording lifted pattern structures from the *domain file*, which is the PDDL file that stores the domain and taken as input by a planner. STN diagrams are used to represent temporal constraints between pattern action pairs, that must exist after their application is complete. The constraints in the STNs of each pattern type are presented to show which actions and constraints need to first be added via search, in order for POPI to infer the other

action and/or constraints that must be added. Some patterns can trigger inference in two different ways; where this is the case and the set of constraints added via search and inference are different between cases, we present an STN diagram for each case. Each pattern type is identified in the left sub-figure, and its corresponding STN(s) in the right sub-figure(s). For the pattern diagrams each rectangle represents an action. The letters above the rectangle denote preconditions, with the start preconditions above on the top left, the invariant (overall) conditions above in the centre and end preconditions above on the top right. The effects of an action are labelled underneath the rectangle, with the start effects of the action on the bottom left and the end preconditions on the bottom right. we can see the format for positioning the preconditions and effects of an action in Figure 3.1.

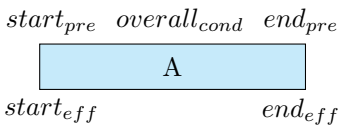



Figure 3.1: Notation of conditions and effects.

Diagrams illustrating the patterns of required concurrency in Figures 3.3 to 3.18 do not display action durations. The proposed pattern detection relies on matching predicates in the conditions and effects lists of the operators. The operators may have fixed or duration inequalities specified. The pattern detection does not consider the duration of the operators as the predicate(s) structure that forms the required concurrency relationship is independent of the duration. It is possible that a pattern with an envelope is detected, where one action must occur entirely within the duration of other and the envelope action is shorter in duration than the inner action. For such a pair of operators to be defined in the domain in this manner would be a modelling error, however our detection does not consider action durations when detecting pattern instances. In this situation, if the actions from this pattern are applied, the STN will detect a temporally inconsistent state during application of actions via inference, the same as would be detected via application of the actions using search.

It is important to note that each of the pattern diagrams presented in this section are each one specific example of each different required concurrency relationship that can exist between two durative actions. In addition, each pattern of actions diagram is presented in its *subset minimal* form, meaning that removing any of the predicates in its example would break the required concurrency relationship being modelled.

### 3.2.1 STN Diagrams and Constraint Interpretations

The STN diagrams for the patterns are each made up of four nodes, two for each action, which are connected with arrows in various formats depending on the constraints between the different endpoints of the actions. The two nodes labelled $A_{\vdash}$ and $A_{\dashv}$ represent the start and end components of action $A$ respectively, the same applies for the nodes labelled $B_{\vdash}$ and $B_{\dashv}$ for action $B$. We now describe the three types of ordering relations that will be used in our STN diagrams to illustrate the different types of constraints. The block arrows in the STN diagrams shown in Figure 3.2a, represent the constraint that any start action, denoted $A_{\vdash}$, must be followed by its corresponding end action, denoted $A_{\dashv}$. This is the relationship between start and end snap actions as presented in Coles et al. [2008]. This Start-End relationship is already maintained in the STN by POPF. The single solid line arrows with block heads illustrated in Figure 3.2b represent the contingent constraint between a concurrent pair of actions, showing that both those snap actions must be in the partial plan constructed so far, with the action being pointed to, occurring after the action being pointed from. We refer to this partial plan as the *plan head*, which is the partial plan from the initial state to the current state. Actions connected by a contingent constraint must be in the plan head in the order shown, for the inference to take place. The broken line arrows with a hollow head, shown in Figure 3.2c depict the constraints which are inferred from knowing the start-end and contingent constraints, using their respective arrow types.

(a) Start-End Constraint.

(b) Contingent Constraint.



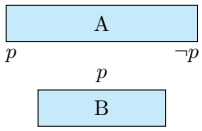
(c) Inferred Constraint.

Figure 3.2: Types of constraints between actions in STNs. First appeared in (Talukdar [2016]).

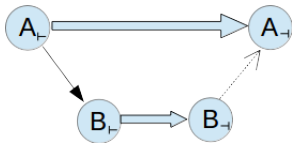
### 3.2.2 Pattern Descriptions

We now move onto describing each of the pattern types that we handle. We will discuss the required concurrency relationship that exists within each pattern and the inferences that can be made, given its detection.

Pattern A in Figure 3.3a displays the situation where one action, A, provides a resource for its duration only. Any action, B, which requires this resource must occur within the temporal window created by action A. In this case action B requires resource P, provided by A for its entire duration, therefore B must occur entirely within the execution of A. As soon as the plan head contains $A_{\vdash}$ and $B_{\vdash}$, with $A_{\vdash}$ before $B_{\vdash}$, and there are no other providers of P, it can be inferred that $B_{\dashv} < A_{\dashv}$.



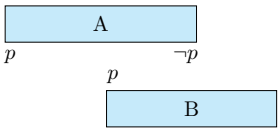

(a) Concurrency Pattern A.



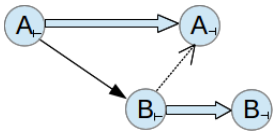

(b) STN Constraints - trigger is A⊢, B⊢.

Figure 3.3: Pattern A. First appeared in (Talukdar [2016]).

Pattern B in Figure 3.4a shows the same temporal window created by A for resource P, however in this case, B only needs the effect of A as a start precondition, therefore B⊢ must occur after A⊢ and before A⊣, but B⊣ can come after A⊣. As soon as A⊢ and B⊢ appear in the plan head, it can be inferred that $B_{\vdash} < A_{\dashv}$.




(a) Concurrency Pattern B.

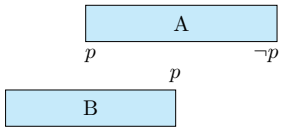



(b) STN Constraints - trigger is A⊢, B⊢.

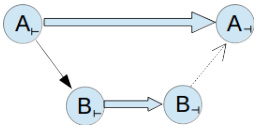
Figure 3.4: Pattern B. First appeared in (Talukdar [2016]).

Pattern C in Figure 3.5a has a similar situation again, except that action B needs action A because action B requires $p$ as an end precondition. However, pattern C is different in that it can be trigger by either A⊢ followed by B⊢, or by B⊢ alone. If the pattern is triggered by B⊢ alone, the planner can do more inference compared to the patterns A and B cases. This is since as soon as B⊢ is in the plan, we can infer that $A_{\vdash} < B_{\dashv}$ and $B_{\dashv} < A_{\dashv}$. If A⊢ and B⊢ trigger the pattern's inference, then more can still be inferred than for pattern B, since it is known that B⊣ must come before A⊣; it is effectively the same inference from pattern A that is achieved. We observe that pattern types A and B are more constrained than pattern C, in terms of the power of the inference. This is since patterns A and B require the start of both actions A and B to be in the plan head to trigger inference, compared with pattern C which can trigger when B⊢ alone is applied, as well as how patterns A and B are triggered.

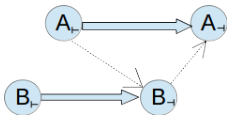
Pattern D in Figure 3.6a represents another scenario where one action must occur entirely within the execution of another action, similar to the situation of concurrency pattern A in Figure 3.3a. However, in this case the reasoning is different, since action A does not create a temporal window for the availability of a resource. Instead action B requires the effect of A⊢, which persists following A⊣. Therefore B⊢ must come after A⊢. However, B⊣ has an effect which is the end precondition for A⊣, thus B⊣ must also come before A⊣. Each action produces




(a) Concurrency Pattern C.



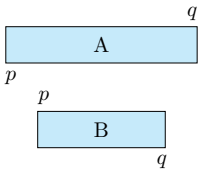

(b) STN Constraints - trigger is A⊢, B⊢.



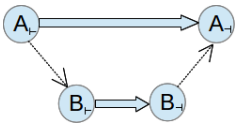

(c) STN Constraints - trigger is B⊢.

Figure 3.5: Pattern C. Adapted from (Talukdar [2016]).

an effect which the other action needs as a precondition. In the case of pattern D, the effect of each action is needed as a precondition of the other action at the same end point. Pattern D is recognised as soon as A⊢ appears in the plan head. This inference is very informative, as we are able to immediately infer $A_{\vdash} < B_{\vdash}$ and $B_{\dashv} < A_{\dashv}$ as the only order of action application for the rest of the pattern.




(a) Concurrency Pattern D.

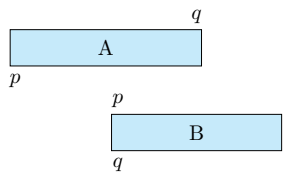



(b) STN Constraints - trigger is A⊢.

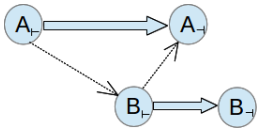
Figure 3.6: Pattern D. First appeared in (Talukdar [2016]).

Pattern E in Figure 3.7a displays a similar situation to pattern D, except the effect that B provides, needed by A⊣, is now provided by B⊢, instead of B⊣. This in effect produces the same type of required concurrency as pattern B, however there is again a different reason for it. B⊢ must occur after A⊢ and before A⊣, but B⊣ can occur after A⊣. This is because the precondition of A⊣ is this time produced by B⊢ instead of B⊣. Again, there is a powerful inference available since the appearance of A⊢ in the plan head would allow the planner to infer $A_{\vdash} < B_{\vdash}$ and $B_{\vdash} < A_{\dashv}$.

Pattern F in Figure 3.8a presents a similar situation as pattern C, except that fact $p$ is needed by B as an end precondition and provides $q$ as its end effect, which action A needs as its end precondition. The temporal constraints in the STN of pattern F in Figure 3.8b can be deduced after the addition of either A⊢ or B⊢ to the plan. As soon as A⊢ or B⊢ appears in the plan head, the inferences of $A_{\vdash} < B_{\dashv}$ and $B_{\dashv} < A_{\dashv}$ are made.

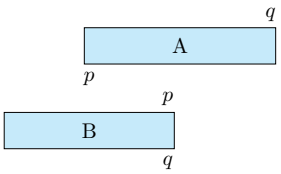



(a) Concurrency Pattern E.

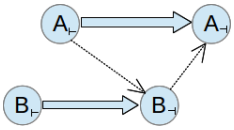



(b) STN Constraints - trigger is A⊢.

Figure 3.7: Pattern E. First appeared in (Talukdar [2016]).



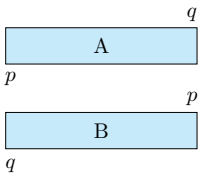

(a) Concurrency Pattern F.



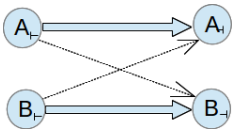

(b) STN Constraints - trigger is A⊢ or B⊢.

Figure 3.8: Pattern F. First appeared in (Talukdar [2016]).

Pattern G presents the case where an end precondition of each action in the pair is provided by the start effect of the other. For this reason, the minimal amount of required concurrency between a pair of actions of this pattern, is where only one end point of both actions, must occur during the execution of the other. The most optimal form of concurrency, in regards to plan makespan, being where one action is executed entirely during the execution of the other. As soon as A⊢ or B⊢ appears in the plan head, A⊢ < B⊣ and B⊢ < A⊣ are inferred. A key point to note about this pattern is that its predicate structure means that, even though there is required concurrency, all four possible orderings of action starts and ends in this pattern can be used when applying them during planning.




(a) Concurrency Pattern G.

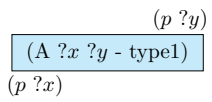



(b) STN Constraints - trigger is A⊢ or B⊢.

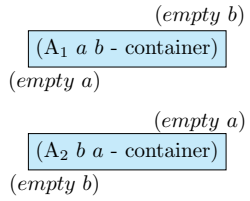
Figure 3.9: Pattern G. First appeared in (Talukdar [2016]).

Pattern $G_{Reflexive}$ has the same temporal constraints as pattern G, and as seen in Figure 3.10b,

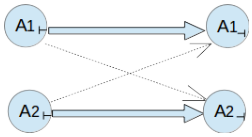
the pattern structure is the same. The difference is that the concurrency is between two instances of the same operator. The symmetric nature of the pattern G structure shown in Figure 3.9a is evident; this is due to the location of the preconditions and effects which means that one instance of a single operator can satisfy the preconditions of a second instance of the same operator. Since pattern facts are assumed to be uniquely achieved by the actions in the pattern structure, only another grounding of the operator can satisfy the end condition of the other one.




(a) Concurrency Pattern $G_{Reflexive}$.




(b) Example of Pattern $G_{Reflexive}$.



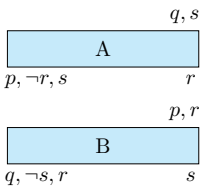

(c) STN Constraints - trigger is $A_{1}\vdash$ or $A_{2}\vdash$.

Figure 3.10: Pattern $G_{Reflexive}$. Adapted from (Talukdar [2016]).

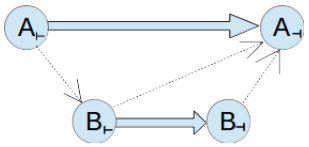
Pattern H is a more restricted version of pattern G, the choices left to the planner are fewer after the pattern has been triggered. Either $A\vdash$ or $B\vdash$ can trigger the inference. If $A\vdash$ is used, the rest of the pattern must be applied as $B\vdash$, $B\dashv$, $A\dashv$. If $B\vdash$ is used, then the pattern is applied as $A\vdash$, $A\dashv$, $B\dashv$. Depending on which pattern action is used to trigger inference, there is only one order in which the remaining actions can be applied, whereas in pattern G there is still a choice of which order to apply the actions ends in, for each trigger case.

Pattern I presents a situation where there is only one possible ordering of the snap actions, where both the starts of actions A and B must both be added via search before required concurrency becomes known and temporal inference can occur. This pattern type triggers in the exact same way as patterns A and B. Pattern I is similar to pattern B, the difference being that $B\dashv$ must come after $A\dashv$, instead of this being a choice. This means that pattern I has a tighter set of constraints and hence the inference that a planner can do using this pattern is slightly stronger than the inference of patten B.

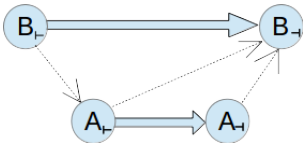
Pattern J presents the situation where $A\vdash$ in the planner implies that action $B$ must also be in the plan. $A\vdash$ being added to the plan via search presents the opportunity for a planner




(a) Concurrency Pattern H.

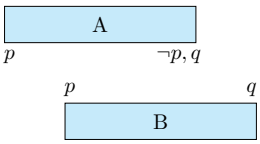



(b) STN Constraints - trigger is A⊢.

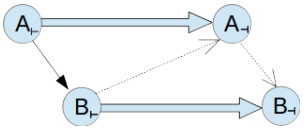



(c) STN Constraints - trigger is B⊢.

Figure 3.11: Pattern H



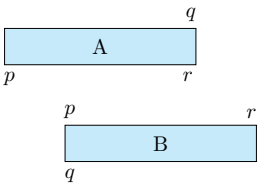

(a) Concurrency Pattern I.



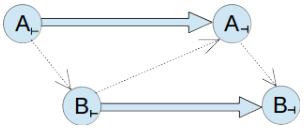

(b) STN Constraints - trigger is A⊢.

Figure 3.12: Pattern I

to add the remaining actions in the only valid remaining order. The order in which the actions must be applied for both pattern types I and J is the same. However, the difference between the patterns is that for pattern I, action A can be applied on its own without action B and therefore a planner cannot trigger inference without applying the start of both actions via search first. This difference means that pattern J is stronger than pattern I because it the structure of pattern J that implies the addition of a new action is needed.




(a) Concurrency Pattern J.

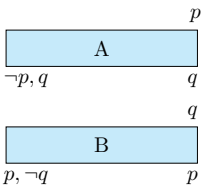



(b) STN Constraints - trigger is A⊢.

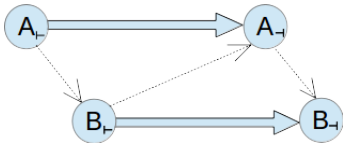
Figure 3.13: Pattern J

Pattern K is similar to pattern H; it is again a more restricted version of pattern G and provides the planner with a stronger inference. Either A⊢ or B⊢ in the plan means that a

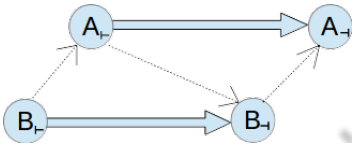
planner can trigger the inference. Following the application of one of these action starts, there is only one sequence in which the remaining pattern actions can be applied. The two possible application sequences see the endpoints of one action interleaved with the other; pattern H is different in that the two sequences see the endpoints points of one action envelope the endpoints of the other.




(a) Concurrency Pattern K.



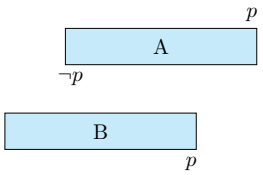

(b) STN Constraints - trigger is A⊢.



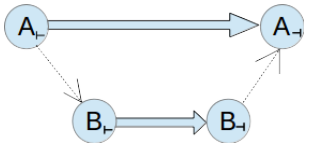

(c) STN Constraints - trigger is B⊢.

Figure 3.14: Pattern K

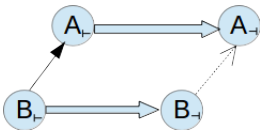
Pattern L is triggered by either A⊢ or B⊢ and A⊢. Both trigger cases enable the inference of a single sequence of application for the remaining actions. However the first case with A⊢ enables inferring that a second action B needs to be included and is therefore more significant.




(a) Concurrency Pattern L.




(b) STN Constraints - trigger is A⊢.

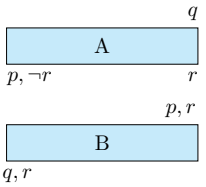



(c) STN Constraints - trigger is B⊢, A⊢.

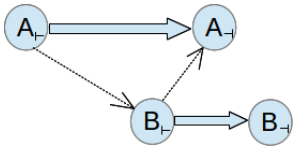
Figure 3.15: Pattern L

Patterns M, N and O all have two trigger cases, the first of which is the same, A⊢. If A⊢ is added to the plan, then the planner can infer that B⊢ must be applied next, but no ordering

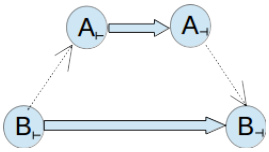
is enforced on how the action endings need to be applied. For patterns M and N, inference can also be triggered using $B_{\vdash}$. For M one remaining order of action application is inferred which is as follows: $A_{\vdash}$, $A_{\dashv}$, $B_{\dashv}$. For N there is also one single sequence for the remaining actions which is $A_{\vdash}$, $B_{\dashv}$, $A_{\dashv}$. For pattern O for its second trigger case, it needs $B_{\vdash}$ followed by $A_{\vdash}$ to enable inference, at which point the ends must be applied as $B_{\dashv}$ followed by $A_{\dashv}$.




(a) Concurrency Pattern M.

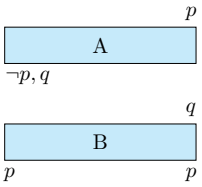



(b) STN Constraints - trigger is $A_{\vdash}$.

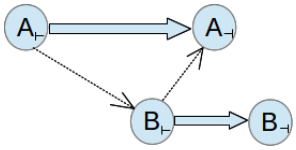



(c) STN Constraints - trigger is $B_{\vdash}$.

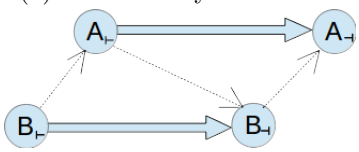
Figure 3.16: Pattern M




(a) Concurrency Pattern N.

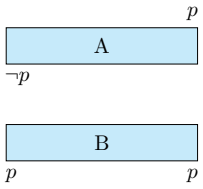



(b) STN Constraints - trigger is $A_{\vdash}$.

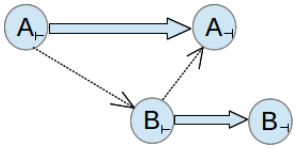



(c) STN Constraints - trigger is $B_{\vdash}$.

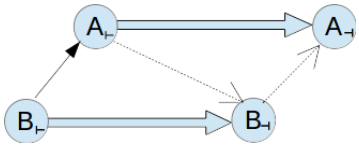
Figure 3.17: Pattern N




(a) Concurrency Pattern O.




(b) STN Constraints - trigger is A⊢.




(c) STN Constraints - trigger is B⊢, A⊢.

Figure 3.18: Pattern O

## 3.3 Inferences from Patterns

In this section we see the inferences made on the basis of each pattern's structure, when it is triggered during planning. Table 3.1 details this information, which is divided into four columns. Column 1 labels which pattern type the tuple is for and in subscript tells us which start action(s) need to be added to the plan and in what order, for the pattern instance to be triggered. For example, the first tuple contains $A_{AB}$ which means that it is referring to pattern type A, where the pattern is triggered by actions A⊢ and B⊢. Column 2 then tells us explicitly that A⊢ must be followed by B⊢ as the trigger actions. The trigger action(s) and the order they must be applied (if there are two trigger actions), is referred to as the *trigger component* of the pattern. Column 3 shows the STN constraints that can be inferred. Some of these constraints are inferred through transitivity, given the inferred constraints above them and are shown brackets. Column 4 tells us which, if any, new actions are added to the plan via the inference. If no new action is added because the starts for both actions in the pattern instance are needed to trigger it and the rest of the pattern consists of only enforcing ordering between the action ends or simply detecting that required concurrency exists, then 'None' appears in that column.

For patterns with only one trigger case, this means that in the forward search framework that we operate in, that this is the only viable trigger case. For some pattern types, there are two trigger cases that enable different inferences. Pattern C is an example of this and is separated into two trigger cases denoted as $C_{AB}$ and $C_B$ informing us of which start action(s) need to be added via search and in what order to enable inference, using the same notation as described for pattern A. $C_{AB}$ is for the case where A⊢ followed by B⊢ appears in the plan head before inference can occur and $C_B$ is for the second case where only B⊢ appears in the plan, but can trigger the pattern for inference on its own. We can see that the amount of gain and

the significance of the inference between these two cases varies, and so does the inferential power. We will see in later discussions how the inference power of each pattern case can be measured and compared.

| Pattern Case | Trigger Action(s) | Constraints Inferred | New Actions Added |
|---|---|---|---|
| $A_{AB}$ | $A_{\vdash} < B_{\vdash}$ | $B_{\dashv} < A_{\dashv}$<br>$(B_{\vdash} < A_{\dashv})$ | None |
| $B_{AB}$ | $A_{\vdash} < B_{\vdash}$ | $B_{\vdash} < A_{\dashv}$ | None |
| $C_{AB}$ | $A_{\vdash} < B_{\vdash}$ | $B_{\dashv} < A_{\dashv}$<br>$(B_{\vdash} < A_{\dashv})$ | None |
| $C_{B}$ | $B_{\vdash}$ | $A_{\vdash} < B_{\dashv}$<br>$B_{\dashv} < A_{\dashv}$<br>$B_{\vdash} < A_{\dashv}$ | $A_{\vdash}$, $A_{\dashv}$ |
| $D_{A}$ | $A_{\vdash}$ | $A_{\vdash} < B_{\vdash}$<br>$B_{\dashv} < A_{\dashv}$<br>$(B_{\vdash} < A_{\dashv})$ | $B_{\vdash}$, $B_{\dashv}$ |
| $E_{A}$ | $A_{\vdash}$ | $A_{\vdash} < B_{\vdash}$<br>$B_{\vdash} < A_{\dashv}$ | $B_{\vdash}$, $B_{\dashv}$ |
| $F_{A}$ | $A_{\vdash}$ | $A_{\vdash} < B_{\dashv}$<br>$B_{\dashv} < A_{\dashv}$<br>$(B_{\vdash} < A_{\dashv})$ | $B_{\vdash}$, $B_{\dashv}$ |
| $F_{B}$ | $B_{\vdash}$ | $A_{\vdash} < B_{\dashv}$<br>$B_{\dashv} < A_{\dashv}$<br>$(B_{\vdash} < A_{\dashv})$ | $A_{\vdash}$, $A_{\dashv}$ |
| $G_{A}$ | $A_{\vdash}$ | $A_{\vdash} < B_{\dashv}$<br>$B_{\vdash} < A_{\dashv}$ | $B_{\vdash}$, $B_{\dashv}$ |
| $G_{B}$ | $B_{\vdash}$ | $A_{\vdash} < B_{\dashv}$<br>$B_{\vdash} < A_{\dashv}$ | $A_{\vdash}$, $A_{\dashv}$ |
| $G_{Reflexive_A1}$ | $A_{1\vdash}$ | $A_{1\vdash} < A_{2\dashv}$<br>$A_{2\vdash} < A_{1\dashv}$ | $A_{2\vdash}$, $A_{2\dashv}$ |
| $G_{Reflexive_A2}$ | $A_{2\vdash}$ | $A_{2\vdash} < A_{1\dashv}$<br>$A_{1\vdash} < A_{2\dashv}$ | $A_{1\vdash}$, $A_{1\dashv}$ |
| $H_{A}$ | $A_{\vdash}$ | $A_{\vdash} < B_{\vdash}$<br>$B_{\dashv} < A_{\dashv}$<br>$(B_{\vdash} < A_{\dashv})$ | $B_{\vdash}$, $B_{\dashv}$ |
| $H_{B}$ | $B_{\vdash}$ | $B_{\vdash} < A_{\vdash}$<br>$A_{\dashv} < B_{\dashv}$ | |

Continued on next page

**Table 3.1 – continued from previous page**

| Pattern Case | Trigger Action(s) | Constraints Inferred | New Actions Added |
|---|---|---|---|
| | | $(A_{\vdash} < B_{\dashv})$ | $A_{\vdash}$, $A_{\dashv}$ |
| $I_{AB}$ | $A_{\vdash} < B_{\vdash}$ | $B_{\vdash} < A_{\dashv}$ | |
| | | $A_{\dashv} < B_{\dashv}$ | None |
| $J_A$ | $A_{\vdash}$ | $A_{\vdash} < B_{\vdash}$ | |
| | | $A_{\dashv} < B_{\dashv}$ | |
| | | $(B_{\vdash} < A_{\dashv})$ | $B_{\vdash}$, $B_{\dashv}$ |
| $K_A$ | $A_{\vdash}$ | $A_{\vdash} < B_{\vdash}$ | |
| | | $B_{\dashv} < A_{\dashv}$ | |
| | | $(B_{\vdash} < A_{\dashv})$ | $B_{\vdash}$, $B_{\dashv}$ |
| $K_B$ | $B_{\vdash}$ | $B_{\vdash} < A_{\vdash}$ | |
| | | $A_{\dashv} < B_{\dashv}$ | |
| | | $(A_{\vdash} < B_{\dashv})$ | $A_{\vdash}$, $A_{\dashv}$ |
| $L_A$ | $A_{\vdash}$ | $A_{\vdash} < B_{\dashv}$ | |
| | | $B_{\dashv} < A_{\dashv}$ | $B_{\vdash}$, $B_{\dashv}$ |
| $L_{BA}$ | $B_{\vdash} < A_{\vdash}$ | $B_{\dashv} < A_{\dashv}$ | None |
| $M_A$ | $A_{\vdash}$ | $A_{\vdash} < B_{\vdash}$ | |
| | | $B_{\vdash} < A_{\dashv}$ | $B_{\vdash}$, $B_{\dashv}$ |
| $M_B$ | $B_{\vdash}$ | $B_{\vdash} < A_{\vdash}$ | |
| | | $A_{\dashv} < B_{\dashv}$ | |
| | | $(A_{\vdash} < B_{\dashv})$ | $A_{\vdash}$, $A_{\dashv}$ |
| $N_A$ | $A_{\vdash}$ | $A_{\vdash} < B_{\vdash}$ | |
| | | $B_{\vdash} < A_{\dashv}$ | $B_{\vdash}$, $B_{\dashv}$ |
| $N_B$ | $B_{\vdash}$ | $B_{\vdash} < A_{\vdash}$ | |
| | | $B_{\dashv} < A_{\dashv}$ | |
| | | $(A_{\vdash} < B_{\dashv})$ | $A_{\vdash}$, $A_{\dashv}$ |
| $O_A$ | $A_{\vdash}$ | $A_{\vdash} < B_{\vdash}$ | |
| | | $B_{\vdash} < A_{\dashv}$ | $B_{\vdash}$, $B_{\dashv}$ |
| $O_{BA}$ | $B_{\vdash} < A_{\vdash}$ | $B_{\dashv} < A_{\dashv}$ | |
| | | $(A_{\vdash} < B_{\dashv})$ | None |

Table 3.1: Pattern triggers and Inferences. Adapted from (Talukdar [2016])

### 3.3.1 Chains of Patterns

In addition to the inferences possible from triggering of a single pattern instance during planning, it is also possible for multiple patterns to chain together enabling all of their inferences to occur. A chain of patterns exists, where two patterns are linked together by a common action that is the inferred action for one pattern and then subsequently the trigger action for another pattern. Let us consider the smallest chain of 2 patterns, an example of which is shown in Figure 3.19. Here, we can see that $A$ triggers the addition of $B$, which in turn triggers the addition of $C$. The actions $A$ and $B$ form a pattern of type D instance and action $B$ and $C$ form a pattern type E instance. A chain of patterns can be made up of the different pattern types as shown in the example in Figure 3.19, or of the same pattern type but constructed in the same way. An example of a large chain of patterns is presented in Figure 3.20, where we see the same situation but there is now a larger set of inferences that are cascaded through, since adding `action_1` to the plan via search, causes the other four actions to be added via inference. The larger the chain, the larger the search to inference ratio becomes. We will discuss chains of patterns and their resulting inferences further in the following chapters.

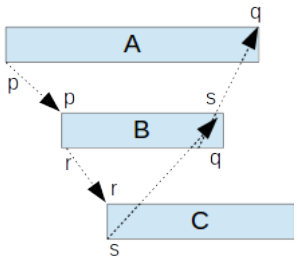



Figure 3.19: Chain of two patterns of different types.

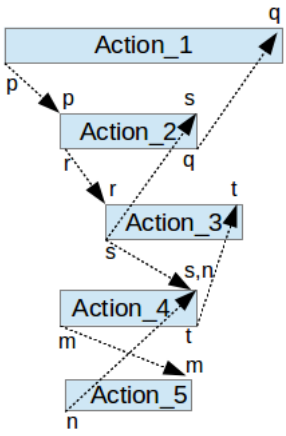



Figure 3.20: Chain of four patterns of different types. First appeared in (Talukdar [2016])

## 3.4 Summary

In this chapter we have provided a catalogue of the pair-wise patterns of required concurrency that we handle. We have presented a set of pattern types that are uniquely different, for pair-wise required concurrency, that are based on the constraints created from the predicate structure of the two actions in each pattern. Additionally, we also have a reflexive pattern type for pattern G, which we have illustrated separately from G, since it is a pattern created from two instances of the same operator. $G_{Reflexive}$ is unique in that the predicate structure is symmetric for both actions, making it possible to have a reflexive version of the pattern. In total this gives us 16 pattern types, of which 15 (counting G and $G_{Reflexive}$ together) have different combinations of flexibility in how the patterns can be triggered and what can be inferred. Chapter 4 will show how the 15 pattern types are uniquely different in structure and exhaustive for all types of pair-wise required concurrency according to the sequences of pattern application.[1] Each of the pattern diagrams illustrated in Section 3.2.2 show one possible way in which the required concurrency relationship it depicts and the inference it enables can exist. However, there are other configurations of predicate structure that can form the same required concurrency relationships enabling the same inferences. In POPI, the pattern types in Section 3.2.2 are the predicate structures that are detected. The predicate names are arbitrary, but the matching of the same predicate configurations as illustrated in each pattern type's diagram is required to detect an instance of that pattern type.

[1] Pattern application is the order the snap actions in the pattern are applied, and after which action required concurrency becomes known to the planner, enabling inference.

# Chapter 4

# Patterns Structures as Sets

## 4.1 Overview

Following the presentation of patterns structures in Chapter 3, for required concurrency between pairs of actions, we now present a formal analysis showing that these patterns and the inferences they enable are complete and exhaustive. In order to achieve this, we consider all the ways in which two actions can occur concurrently and enable opportunities for doing inference.

In this chapter we present a formal analysis of required concurrency between action pairs. The analysis presents patterns of required concurrency as sets of sequences, where each sequence in a set represents a viable order in which the snap actions can be applied during state progression, without violating the constraints of the required concurrency. The *trigger point* of a pattern sequence is the point at which required concurrency becomes known to the planner, enabling it to make inferences. This examination of pattern structures sees the actions and the trigger point being coupled together and analyses the combinations of the possible concurrent orderings, considering the trigger point as a crucial component of the pattern. This is to determine exactly what unique pattern structures exist, where each type of structure is distinguished and represented by a unique set of pattern sequences. Each individual sequence is a valid pattern structure on its own, meaning that the set for it only contains that one sequence of application. When considering structures where an action pair must be concurrent, but less restricted in the ordering of the snap actions, this corresponds to sets containing multiple sequences.

This perspective gives us a broader and more comprehensive method of recognising required concurrency and a basis for provably determining that we cover a complete set of pattern types for pairs of durative actions. The terms *pattern ordering* and *pattern sequence* are used interchangeably from here onwards and both refer to the order in which the snap

actions can be applied, and the point at which inference can occur. Each 'Pattern type' refers to a unique set of these pattern sequences, representing the overall relationship between two concurrent actions, where there exists one or more pattern sequences.

## 4.2 Pattern Sequences

This section introduces all of the individual concurrent pattern orderings possible for a pair of durative actions $A$ and $B$. The trigger point at which required concurrency becomes known to the planner is considered key information, since it determines when the planner can do inference. The location of the trigger point in the sequence also tells the planner exactly which actions can be inferred and need to be applied, given the action choices made so far. The asterisk '*' symbol is used to represent the trigger point in a pattern sequence. The snap action(s) before the trigger point will be referred to as the *sequence head*. The sequence head contains important information, in that it tells us whether an action $A$ that starts first in a sequence can appear in a plan on its own; meaning there is no dependency on the second action $B$ in the pattern. The remaining snap actions that come after the trigger point in a sequence are referred to as the *sequence tail*.

To fully illustrate what the difference between these two scenarios are, let us distinguish the two cases. The pattern sequence $A_{\vdash}$, $B_{\vdash}$, * , $B_{\dashv}$, $A_{\dashv}$ allows for the execution of both actions $A$ and $B$ as follows, [$A_{\vdash}$, $B_{\vdash}$, $B_{\dashv}$, $A_{\dashv}$], but it also allows for just $A$ to execute on its own without $B$ as follows [$A_{\vdash}$, $A_{\dashv}$]. This is because there is only a one way resource production and consumption relationship between $A$ and $B$, where $B$ depends on $A$ but not vice versa. Conversely, now when we consider the same sequence of actions, but with the trigger point moved to be after $A_{\vdash}$, as follows, $A_{\vdash}$, * , $B_{\vdash}$ , $B_{\dashv}$, $A_{\dashv}$, there is now a two way dependency between $A$ and $B$, where $A$ cannot occur on its own as in the first case. Only the concurrent sequence [$A_{\vdash}$, $B_{\vdash}$, $B_{\dashv}$, $A_{\dashv}$] can be applied, which means required concurrency is known to exist immediately after applying only $A_{\vdash}$, without needing to see $B_{\vdash}$ as well, as is the situation in the first case. This information, along with the details of the specific constraints between the actions, that form the required concurrency relationship, determines which type of pattern it is. Each unique set of pattern sequences makes up a different pattern structure, the strictest cases of which are the sets containing a single sequence.

When considering patterns of durative action pairs, this translates to four snap action end points (happenings). A pattern triggers and enables temporal inference, after either the first or second snap action has been applied from the sequence. The sub-sequence of actions that trigger inference correspond to the actions in the trigger component of a pattern, defined in Section 3.3 of Chapter 3. A different trigger component means a different pattern sequence

for a pattern structure, since the number and order of the actions in the trigger component matters and also constitutes an entirely different pattern sequence. Conversely, two sequences with the same trigger component does not necessarily mean that they are part of same pattern sequence, since the ordering of the sequence tail (end snap actions) may be different.

There are four snap actions in a single pattern sequence; the trigger point is after either the start of action $A$ or after the start of $A$ and $B$. This means that there are four different patterns sequences starting with $A$. Since $A$ and $B$ are symmetric, there are another four starting with $B$, with trigger points in the same places. Therefore there are eight sequence to consider in total. When looking at each pattern sequence individually, each sequence that starts with an action $A$ has a symmetric counterpart starting with an action $B$. In general, a pattern structure cannot have in its set of sequences, a pair of symmetric sequences. However, there is an exception to this generalisation which will be explained later in the chapter, but is a case ruled out in this analysis, which we will discuss.

Table 4.1 presents the eight sequences, showing all the combinations in which a pair of durative actions may be applied in forward search and the different points at which required concurrency becomes known and inference can occur. It is important to note that the column showing the constraints inferred, refer specifically to those sets containing these single sequences. The constraints that are in brackets are those that are inferred through transitivity, having inferred the other non-bracketed constraints. In addition, sequences 1, 2, 3 and 4 are symmetric to sequences 5, 6, 7 and 8 respectively. This being the case, it is still important to enumerate all 8 of them, since we consider the cross space combination of all 8 sequences for the purpose of considering pattern sets where there are multiple sequences. Although not all of these sets are valid pattern sets, as will be described in Section 4.3, some are, and all combinations must be considered in order for this analysis to be complete.

| Sequence No. | Pattern Sequence | Constraints Inferred | New Actions added to Plan |
|---|---|---|---|
| 1. | $A_\vdash$, $B_\vdash$, * , $B_\dashv$, $A_\dashv$ | $B_\dashv < A_\dashv$<br>$(B_\vdash < A_\dashv)$ | None |
| 2. | $A_\vdash$, * , $B_\vdash$, $B_\dashv$, $A_\dashv$ | $A_\vdash < B_\vdash$<br>$B_\dashv < A_\dashv$<br>$(B_\vdash < A_\dashv)$<br>$(A_\vdash < B_\dashv)$ | B |
| 3. | $A_\vdash$, $B_\vdash$, * , $A_\dashv$, $B_\dashv$ | $A_\dashv < B_\dashv$<br>$(A_\vdash < B_\dashv)$ | None |
| 4. | $A_\vdash$, * , $B_\vdash$, $A_\dashv$, $B_\dashv$ | $A_\vdash < B_\vdash$<br>$A_\dashv < B_\dashv$<br>$(B_\vdash < A_\dashv)$<br>$(A_\vdash < B_\dashv)$ | B |
| 5. | $B_\vdash$, $A_\vdash$, * , $A_\dashv$, $B_\dashv$ | $A_\dashv < B_\dashv$<br>$(A_\vdash < B_\dashv)$ | None |
| 6. | $B_\vdash$, * , $A_\vdash$, $A_\dashv$, $B_\dashv$ | $B_\vdash < A_\vdash$<br>$A_\dashv < B_\dashv$<br>$(A_\vdash < B_\dashv)$<br>$(B_\vdash < A_\dashv)$ | A |
| 7. | $B_\vdash$, $A_\vdash$, * , $B_\dashv$, $A_\dashv$ | $B_\dashv < A_\dashv$<br>$(B_\vdash < A_\dashv)$ | None |
| 8. | $B_\vdash$, * , $A_\vdash$, $B_\dashv$, $A_\dashv$ | $B_\vdash < A_\vdash$<br>$B_\dashv < A_\dashv$<br>$(A_\vdash < B_\dashv)$<br>$(B_\vdash < A_\dashv)$ | A |

Table 4.1: Patterns of Actions with Required Concurrency. Asterisk '*' symbol comes after the subset of snap actions required to be in plan before required is known to exist by the planner. Constraints in brackets are inferred through transitivity, having inferred the other constraints.

## 4.3 Sets of Pattern Sequences

It is possible for actions in a pattern structure to be applied in one of multiple different sequences, if the trigger component is different or if the trigger component is the same but there is no constraint for order in which the ends of the actions are applied. We have observed in Section 4.2 that all of the eight individual sequences are possible as pattern structures in their own right, and each sequence has a symmetric counterpart. We now look at how to view the pattern structures, for sets containing multiple sequences. We provide theorems and subsequent proofs to show which sets of sequences constitute valid pattern structures with required concurrency. In order to calculate the number of valid sets of pattern sequences, we first must calculate all of the possible combinations of pattern sequences. As there is no ordering or repetition of the sequences in a set, having eight elemental sequences means that

there are 255 distinct set combinations, excluding the empty set. For a pattern structure, where its set contains multiple sequences, it must be that any of the sequences in the set can be applied. The pattern sets which exist will be determined with the following assumptions in place:

1. All actions are durative actions using PDDL 2.1 (Fox and Long [2003]).
2. No conditional effects are used.
3. The existence of a pattern structure cannot depend on the ability of the planner to schedule simultaneous happenings.
4. Disjunctive preconditions are not used.
5. Negative preconditions are not used.
6. None of the 'pattern structure facts' are true in the trigger state (state where the trigger action becomes applicable). These are the facts that maintain the specific type of required concurrency relationship between two actions, A and B, of a distinct pattern type (corresponding to a unique pattern set).
7. The pattern facts, can only be achieved by the two actions in the pattern structure.

Initially it may seem counter intuitive to combine the individual pattern sequences in Table 4.1 together into a pattern structure. However, it is possible for patterns of required concurrency to exist with varying levels of constraints. The goal of combining multiple sequences is to determine all of the pair-wise patterns that exist, that have varying levels of constraints.

An important step in looking at how patterns of required concurrency can be constructed by the combining different individual sequences, is to first define exactly what it means when there is more than one sequence in a set. When there are multiple sequences in a set, it must be possible for the planner, under some circumstance, to be able to apply each of the actions in the pattern and trigger inference in the order defined in each sequence. The pattern structure must still enforce required concurrency with no sequential application of the actions possible. Effectively, the group of pattern structures created from combining multiple sequences within a set, is a cross breed of pattern sets, where some are stronger and more informative than the individual sequence sets, and some are weaker, depending on the comparison being made.

To illustrate the concept with an example, we can see trivially that in the case where an action $B$ only needs at its start, a temporarily available resource provided by an action $A$, that required concurrency still exists regardless of the order in which the end of $A$ and $B$ are applied. This is a combination of sequence 1 and 3, where it must be possible to apply both

of them. This is exactly the situation seen in pattern B, shown in Figure 3.4 in Chapter 3. In this type of situation, this means that the start of A and B must be seen before required concurrency becomes known to the planner and inference can occur, however there is also now still some choice about which order to apply the ends of the two actions. This is unlike the set where there is only sequence 1 or only sequence 3, where we have a disjunction of these sequences and the planner knows a particular order must be enforced. In the combined case of 1 and 3, there is a conjunction of sequences with the same sequence head and different sequence tails. The planner can only infer that there is in fact required concurrency that exists and must be maintained, but cannot infer the order of applying the end actions. It is because of this, that when there is a conjunction of two sequences with the same sequence head, but different sequence tails in the same set, we view the combination of these sequences as effectively being one sequence, but with choice about which sequence tail to apply. The pairs of sequence that this applies to, in addition to 1-3 produced from set {1, 3}, are {2, 4} to produce 2-4, {5, 7} to produce 5-7 and {6, 8} to produce 6-8. The sequences 1-3 and 5-7 are symmetric, as are sequences 2-4 and 6-8. On the surface, viewing these particular pairs of pattern sequences in this manner does not seem to add any value, however the usefulness of this will become clear in the proof for Theorem 4.3. It is important to note that Table 4.2 is made up of the same individual pattern sequences in Table 4.1, but also includes a different representation of pairs of sequences which have the same sequence heads but different sequence tails for the sets in which these sequences appear together.

In general, when there is more than one sequence in a set, the number of constraints on the order of action application is reduced, as there may be some choice about the order of application, depending on the number of sequences in the set and if there is a pair of sequences with the same trigger component. This will become clearer in Section 4.4 which discusses pattern strength and the inferential power of each pattern structure according to its trigger component.

**Definition 4.1** (Pattern Set)**.** A subset of the 8 pattern sequences presented in Table 4.2.

**Definition 4.2** (Sequence Symmetry)**.** Exists between two different sequences when the positions of the two start snap actions and the two end snaps actions are in the same positions of their sequences, the trigger point is in the same location, but the actions in those positions between the sequences are different.

**Definition 4.3** (Set Symmetry)**.** Exists between two sets when each set contains the symmetric sequences of the other set.

**Definition 4.4** (Contradiction)**.** Exists in a pattern set, if for any pair of its sequences, a pair of actions $A$ and $B$ cannot be constructed with required concurrency, such that the planner can execute **either** of its sequences.

| Sequence No. | Pattern Sequence | Combined Meaning |
|---|---|---|
| 1. | $A_{\vdash}$, $B_{\vdash}$, * , $B_{\dashv}$, $A_{\dashv}$ | N/A |
| 2. | $A_{\vdash}$, * , $B_{\vdash}$, $B_{\dashv}$, $A_{\dashv}$ | N/A |
| 3. | $A_{\vdash}$, $B_{\vdash}$, * , $A_{\dashv}$, $B_{\dashv}$ | N/A |
| 4. | $A_{\vdash}$, * , $B_{\vdash}$, $A_{\dashv}$, $B_{\dashv}$ | N/A |
| 5. | $B_{\vdash}$, $A_{\vdash}$, * , $A_{\dashv}$, $B_{\dashv}$ | N/A |
| 6. | $B_{\vdash}$, * , $A_{\vdash}$, $A_{\dashv}$, $B_{\dashv}$ | N/A |
| 7. | $B_{\vdash}$, $A_{\vdash}$, * , $B_{\dashv}$, $A_{\dashv}$ | N/A |
| 8. | $B_{\vdash}$, * , $A_{\vdash}$, $B_{\dashv}$, $A_{\dashv}$ | N/A |
| 1-3. | $A_{\vdash}$, $B_{\vdash}$, * , [$B_{\dashv}$, $A_{\dashv}$] [$A_{\dashv}$, $B_{\dashv}$] | No Inference Possible |
| 2-4. | $A_{\vdash}$, * , $B_{\vdash}$, [$B_{\dashv}$, $A_{\dashv}$] [$A_{\dashv}$, $B_{\dashv}$] | Infer addition of $B$, but not ordering of $A_{\dashv}$ and $B_{\dashv}$ |
| 5-7. | $B_{\vdash}$, $A_{\vdash}$, * , $A_{\dashv}$, $B_{\dashv}$ [$B_{\dashv}$, $A_{\dashv}$] | No Inference Possible |
| 6-8. | $B_{\vdash}$, * , $A_{\vdash}$, [$A_{\dashv}$, $B_{\dashv}$] [$B_{\dashv}$, $A_{\dashv}$] | Infer addition of $A$, but not ordering of $A_{\dashv}$ and $B_{\dashv}$ |

Table 4.2: Pattern Action Sequences, displaying the representation sequence pairs that have the same sequence head, but different sequence tails, when they appear together in a pattern set.

**Definition 4.5** (Sequence Specific)**.** The property of a Pattern Set, if the sequences specified are the only orderings in which the pattern actions and their associated inferences can occur and no others.

**Definition 4.6** (Invalid Pattern Set)**.** A pattern set that contains one or more contradictions.

**Theorem 4.1.** *All sets that contain a sequence starting with* {$A_{\vdash}$, $B_{\vdash}$, * ... } *and another sequence starting with* {$A_{\vdash}$, * , $B_{\vdash}$ ... } *contain a contradiction and are not valid pattern sets. These are the sets containing the sets* {*1, 2*}, {*3, 4*}, {*5, 6*}, {*7, 8*}, {*1,4*}, {*2, 3*}, {*5, 8*}, {*6, 7*} *as subsets.*

*Proof.* First, consider the set {1, 2}. Assume that the set {1, 2} is a valid pattern; this corresponds to $A_{\vdash}$, $B_{\vdash}$, * , $B_{\dashv}$, $A_{\dashv}$ and $A_{\vdash}$, * , $B_{\vdash}$ , $B_{\dashv}$, $A_{\dashv}$ for sequences 1 and 2 respectively.

Let us analyse what each of these pattern sequences mean in turn. For sequence 1, the sequence representation shows us that action A can occur on its own and exist in the plan without any dependency on action B. This is indicated by the fact that the trigger point '*', when required concurrency becomes known and the time when inference can occur, is after the start of both actions A and B. In contrast, action B cannot exist without action A and must exist within the envelope of $A_{\vdash}$ and $A_{\dashv}$. For sequence 2, we now see that the trigger point is immediately after the start of action A and the rest of the pattern sequence is the same as sequence 1. This means that the point at which required concurrency becomes

known, is after $A_{\vdash}$ alone and simultaneously after both $A_{\vdash}$ followed by $B_{\vdash}$, have been applied. This is a contradiction, since the earliest time point at which the required concurrency of $A$ and $B$ becomes known is always the same, when the start of the two actions is in the same order for two different pattern sequences. It is not logical that only $A_{\vdash}$ and simultaneously $A_{\vdash}$ followed by $B_{\vdash}$, are both possible as being the earliest time at which the required concurrency for the action pair becomes known, when the order of the snap actions are the same. It is clear that it is the two start actions and their ordering that is important. Therefore the same reasoning applies for why sets containing {3, 4} cannot be valid patterns, since the set {1, 2} is the same as {3, 4}, except that for sequence 3 and 4, the end of action A comes before the end of action B.

Again, the same reasoning is true for sets containing {5, 6}, {7, 8} as subsets, which we can see by switching around the labels of actions A and B for these sets. This is since {1, 2} and {5, 6} are symmetric to each other, as are the sets {3, 4} and {7, 8}.

We have seen that the reason for the contradiction in sets containing {1, 2}, {3, 4}, {5, 6}, {7, 8} as subsets, is that the two sequences in each of these sets, order the two start actions the same, but trigger the inference at two different locations, which has been shown as not possible. This is a general rule which is also applicable for the sets containing {1, 4}, {2, 3}, {5, 8}, {6, 7} as subsets. These pairs of sequences consist of the contradictory starting components {$A_{\vdash}$, $B_{\vdash}$, * , ... } and {$A_{\vdash}$, * , $B_{\vdash}$ , ... } which cannot exist together for a pair of actions A and B in a pattern of required concurrency. □

**Theorem 4.2.** *Sets containing sequences that have both {$A_{\vdash}$, $B_{\vdash}$ * ... } and {$B_{\vdash}$, $A_{\vdash}$ * ... } as the trigger component are not valid pattern sets. Therefore the following pattern sets are not valid: {1, 5}, {1, 7}, {3, 5}, {3, 7}.*

*Proof.* Any set which contains sequences starting with both {$A_{\vdash}$, $B_{\vdash}$ * ... } and {$B_{\vdash}$, $A_{\vdash}$ * ... } as sub-components results in a pattern where actions A and B can be sequential, hence the concurrency is not required. This is since if $A_{\vdash}$ and $B_{\vdash}$ are needed in both orders to trigger inference, this means that neither action can trigger inference on its own, because it must be that neither action needs the other to occur concurrently in order to execute successfully. This must mean that actions A and B can each appear in the plan without the addition of the other or the actions can appear sequentially. However, it is prudent to acknowledge that pattern structures for the sets containing these sequences may still exist, but that the concurrency of the actions is optional and not required and is therefore ruled out. An example of such a situation can be seen with set {1, 5}. Figure 4.1 shows an instantiation of the structure where only sequences {1, 5} are possible for the concurrent occurrence of the actions, where this is a non-required concurrent pair of actions. Without being able to schedule simultaneous happenings without epsilon time gaps, this pattern cannot exist with required concurrency.

□

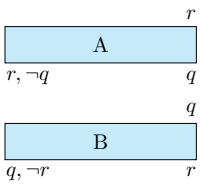



Figure 4.1: Non-required Concurrency for set {1, 5}.

**Theorem 4.3.** *The following sets are not valid pattern sets:*

$$\{1,\ 6\},\ \{1\text{-}3,\ 6\},\ \{1,\ 6\text{-}8\},\ \{1\text{-}3,\ 6\text{-}8\},\ \{3,\ 8\},\ \{1\text{-}3,\ 8\},$$

$$\{2,\ 5\},\ \{2,\ 5\text{-}7\},\ \{2\text{-}4,\ 5\},\ \{2\text{-}4,\ 5\text{-}7\},\ \{4,\ 7\},\ \{4,\ 5\text{-}7\}.$$

*Proof.* To start with, let us consider the sets: {1, 6}, {1-3, 6}, {1, 6-8}, {1-3, 6-8}, {3, 8}, {1-3, 8}, where for each set there exists the trigger components {A⊢, B⊢, * ... } and {B⊢, * ... }. In this situation, action $A$ can exist on its own, but action $B$ cannot. Both actions are able to start independently of one another. This means that for required concurrency to exist, it must be that it is B⊣ specifically that has a precondition achieved as an effect of $A$, since we know $A$ can exist on its own and therefore cannot have any conditions that depend on $B$. Under these circumstances, there are four possibilities in which a fact $p$ needed by B⊣ can be achieved and deleted and still be true in time for B⊣ to be applied. We refer to these four possibilities as *resolution cases*, labelled $w$, $x$, $y$, $z$, shown in Table 4.3. For any of these pattern sets to be possible, there must be a common resolution case among all of the sequences in the set, in order to satisfy the precondition of B⊣. Table 4.4 shows which pattern resolution case(s) are available for each component sequence in the sets being considered. For each of the sets: {1, 6}, {1-3, 6}, {1, 6-8}, {1-3, 6-8}, {3, 8}, {1-3, 8}, none of the sequences in the set have a common resolution case, showing that they are not viable sets.

Now, let us consider the sets: {2, 5}, {2, 5-7}, {2-4, 5}, {2-4, 5-7}, {4, 7}, {4, 5-7}. As these sets are symmetric to the first six sets asserted as not being viable, the same reasoning follows for why they are not possible, when we consider that the same criteria must be satisfied for sets containing the symmetric trigger components {B⊢, A⊢, * ... } and {A⊢, * ... }. There must be a common resolution case for satisfying the precondition $p$ of A⊣ provided by $B$.

□

| Resolution Case. | Pre($B_\dashv$) Achiever(s) | Pre($B_\dashv$) Deleter(s) |
|---|---|---|
| $w$ | $A_\vdash$ | $A_\dashv$ |
| $x$ | $A_\vdash$ | $B_\vdash$ |
| $y$ | $A_\dashv$ | $B_\vdash$ |
| $z$ | $A_\vdash$, $A_\dashv$ | $B_\vdash$ |

Table 4.3: This table shows the 4 resolutions cases, that each of the sequences that are members of the pattern sets in Theorem 4.3 must have in common, in order to be a valid pattern set.

| Sequence No. | Pattern Structure | Pattern Resolution Case(s) |
|---|---|---|
| 1. | $A_\vdash$, $B_\vdash$, * , $B_\dashv$, $A_\dashv$ | w |
| 3. | $A_\vdash$, $B_\vdash$, * , $A_\dashv$, $B_\dashv$ | y, z |
| 6. | $B_\vdash$, * , $A_\vdash$, $A_\dashv$, $B_\dashv$ | x, y |
| 8. | $B_\vdash$, * , $A_\vdash$, $B_\dashv$, $A_\dashv$ | w, x |
| 1-3. | $A_\vdash$, $B_\vdash$, * , [$B_\dashv$, $A_\dashv$] [$A_\dashv$, $B_\dashv$] | No Resolution Case |
| 6-8. | $B_\vdash$, * , $A_\vdash$, [$A_\dashv$, $B_\dashv$] [$B_\dashv$, $A_\dashv$] | x, z |

Table 4.4: This table shows the 4 resolutions cases, that each sequence which is a member of one of the pattern sets in Theorem 4.3 must have in common, in order to be a valid pattern set with required concurrency.

**Lemma 4.1.** *The following pairs of pattern sets are symmetric: ({1},{5}), ({2},{6}), ({3},{7}), ({4},{8}), ({1-3},{5-7}), ({1, 8},{4, 5}), ({2-4},{6-8}), ({2, 8},{4, 6}), ({2-4, 6-8}, {2-4, 6-8}), ({2, 6},{2, 6}), ({4, 8},{4, 8}), ({2, 7},{3, 6}), ({2-4, 6},{2, 6-8}), ({2-4, 8},{4, 6-8}), ({2-4, 7},{3, 6-8}).*

*Proof.* Section 4.2 mentioned that the individual sequences 1, 2, 3 and 4 are symmetric with sequences 5, 6, 7 and 8 and respectively. In addition, this Section 4.3 mentions that 1-3 is symmetric with 5-7, as are 2-4 and 6-8. This is because by switching the locations of actions $A$ and $B$ we attain a group of sequences beginning with $B$, that have the same interleaved interactions and are symmetric in structure to the sequences starting with $A$. The pairs of symmetric sets are identified by using Algorithm 2. This algorithm takes one set from each pair of sets mentioned and performs a function which maps each sequence in the set to its symmetric sequence, for all of its sequences and then outputs the set which is symmetric to the one given as input; the symmetric sets being the other component of the pair, that the input set was chosen from. Let us consider the first pair of sets, where there is one sequence in each set. If we input as $S$ the set $\{1\}$ into Algorithm 2, we see that x = 1 and since 1 is less than 5, x is increased by 4, giving it a value of 5. This is then added to set $S'$; since there are no more sequences in $S$, $S'$ is returned containing only the sequence 5. The set $\{5\}$ is exactly the set which is symmetric to $\{1\}$. If we input $\{5\}$, this translates back to $\{1\}$,

since 5 - 4 = 1. Let us now consider the pair ({1-3},{5-7}). Even though when 1 and 3 are together in the same set, we view them as one conjunctive sequence 1-3, as we also do for 2 and 4, 5 and 7, 6 and 8, we still apply the function of Algorithm 2, for each of its constituent sequences, in this case 1 and 3, separately. We then join the two symmetric sequences into one conjunctive sequence again. Therefore, when we input {1-3} into Algorithm 2, we perform the operations: 1 + 4 = 5 and 3 + 4 = 7, join these two resultant sequences together and get the set {5, 7}. We see that {5-7} is exactly the set listed as being symmetric to {1-3}. Inputting {5-7} translates back to {1-3}, since 5 - 4 = 1 and 7 - 4 = 3.

This same process applies to all of the pairs of sets mentioned, where inputting one of its sets produces its symmetric counterpart set. We can verify that the mappings of these pattern sequence numbers are correct by referring back to Table 4.2 that shows all of the pattern sequence structures. The first eight tuples show the individual sequences and we can see that for each sequence, its symmetric counterpart is 4 tuples away, hence the reason for the function in Algorithm 2 being the way it is. We should also note that the sets {2, 6}, {4, 8} and {2, 4, 6, 8} are self symmetric, since when input each of these three sets into Algorithm 2, we get the same sets as the output.

□

**Algorithm 2:** identifySymmetricSets

```
   Input  : patternSet S
   Output : patternSet S'
1  foreach sequenceNumber x in S do
2      if x < 5 then
3          x = x + 4;
4          S'.add(x);
5      else
6          x = x - 4;
7          S'.add(x);
8  return S'
```

$$f(x) = \begin{cases} x + 4, & \text{if } x < 5 \\ x - 4, & \text{otherwise} \end{cases} \tag{4.1}$$

**Definition 4.7.** $S = S'$ if and only if $S' = \{\ f(x) : x \in S\ \}$

**Theorem 4.4.** *The following pattern sets constitute the only valid pattern sets, when excluding symmetries: {1}, {2}, {3}, {4}, {1-3}, {1, 8}, {2-4}, {2, 8}, {2-4, 6-8}, {2, 6}, {4, 8}, {2, 7}, {2-4, 6}, {2-4, 8}, {2-4, 7}*

*Proof.* Having proved Theorems 4.1, 4.2, 4.3 and Lemma 4.1, we see that only these sets constitute the unique pattern structures. This is reflected in Table 4.5, which shows by

pattern label one tuple per unique pattern set. Reflexive patterns are not included in this, hence $G_{Reflexive}$ is not included in Table 4.5.

□

| Pattern Label. | Set | Symmetric Set |
|---|---|---|
| A | {1} | {5} |
| B | {1-3} | {5-7} |
| C | {1, 8} | {4, 5} |
| D | {2} | {6} |
| E | {2-4} | {6-8} |
| F | {2, 8} | {4, 6} |
| G | {2-4, 6-8} | {2-4, 6-8} |
| H | {2, 6} | {2, 6} |
| I | {3} | {7} |
| J | {4} | {8} |
| K | {4, 8} | {4, 8} |
| L | {2, 7} | {3, 6} |
| M | {2-4, 6} | {2, 6-8} |
| N | {2-4, 8} | {4, 6-8} |
| O | {2-4, 7} | {3, 6-8} |

Table 4.5: Valid Pattern Sequence Sets containing required concurrency.

Symmetric sequences can both appear in the same set of sequences as a valid pattern structure, if both sequences can be achieved without a contradiction. This is the case for sets {2, 6}, {4, 8} and {2-4, 6-8}. Sets {2, 6}, {4, 8} and {2-4, 6-8} being self symmetric relies on Lemma 4.1 and justifies Equation 4.1 and Algorithm 2. Equation 4.1 is the function encoded in Algorithm 2. Therefore, Algorithm 2 is one representation of the result of Equation 4.1. Using Algorithm 2 as an operator for identifying symmetric sets, we can see that inputting the sets {2, 6}, {4, 8} and {2-4, 6-8} into the algorithm produces the same sets of sequences as the output.

Given the unique pattern structures that have been identified, we can put the patterns into categories, according to the combination of sequences in each pattern's associated set. We identify three categories of patterns. One category is for patterns where each sequence in its set has two trigger actions. Another category is for patterns where each sequence in its associated set has one trigger action. The last category is for patterns where there is both a one trigger action sequence and a two trigger action sequence in its set. All odd numbered sequences have a two action trigger component and all even numbered sequences have a one action trigger component. Given this criteria for categorising the patterns, in the two trigger action category we have patterns A, B and I. In the single trigger action category, we have

patterns D, E, F, G, H, J, K, M and N. In the mixed category with one and two trigger actions sequences contained in their sets, we have patterns C, L and O.

## 4.4 Pattern Strength

We now discuss how to determine the strength of each pattern set shown in Table 4.5, according to each of its distinct trigger component of actions displayed in Table 4.6. The key features in determining the inferential power of each pattern type, given each of its distinct trigger action sets are:

1. The size of the trigger component either being 1 or 2 snap actions.
2. The number of different trigger action components.
3. The number of different orders in which the remaining 2 or 3 snap actions can be applied, according to the required concurrency constraints, given the trigger action(s) already applied.
4. The location of the trigger point in each pattern sequence.

In Table 4.6, where there is more than one set of trigger actions for a pattern type, we distinguish these in separate tuples, such that it is clear how many endpoints orderings the planner can choose from as the result of using each trigger component for each pattern type. Patterns C, F, G, H, K, L, M, N and O are pattern types where this is the case. For these pattern types, each of its trigger components are included in subscript in the pattern name, so that they can be referred to individually.

We determine the inference power of each of the cases in Table 4.6, according to the number of choices left in the ordering of the remaining snap actions (endpoints), given the specific trigger component applied for that case. The calculation for this inference power is based on the size of the trigger component and the number of ordering choices left after applying that set of trigger actions. Table 4.7 presents the 4 possible cases. Pattern cases where the trigger component size is 2, meaning both actions are needed in the plan before inferring anything, with 2 orderings of the action ends possible, have the lowest inferential power of 1. The same situation with only 1 possible ordering after applying the trigger component, is measured with a power of 2. The two cases where only 1 start action is needed have the highest powers. If 2 orderings of the remaining 3 snap actions are possible after applying the trigger component, this has a power of 3. The most inferential power comes from cases with only 1 possible ordering for all 3 remaining actions after applying a set of 1 trigger action, resulting in a power value of 4.

| Pattern Trigger Case | Trigger Action(s) | Trigger size | no. Choices | Power |
|---|---|---|---|---|
| $A_{AB}$ | $A_{\vdash}$, $B_{\vdash}$ | 2 | 1 | 2 |
| $B_{AB}$ | $A_{\vdash}$, $B_{\vdash}$ | 2 | 2 | 1 |
| $C_{AB}$ | $A_{\vdash}$, $B_{\vdash}$ | 2 | 1 | 2 |
| $C_B$ | $B_{\vdash}$ | 1 | 1 | 4 |
| $D_A$ | $A_{\vdash}$ | 1 | 1 | 4 |
| $E_A$ | $A_{\vdash}$ | 1 | 2 | 3 |
| $F_A$ | $A_{\vdash}$ | 1 | 1 | 4 |
| $F_B$ | $B_{\vdash}$ | 1 | 1 | 4 |
| $G_A$ | $A_{\vdash}$ | 1 | 2 | 3 |
| $G_B$ | $B_{\vdash}$ | 1 | 2 | 3 |
| $G_{ReflexiveA1}$ | $A_{1\vdash}$ | 1 | 2 | 3 |
| $G_{ReflexiveA2}$ | $A_{2\vdash}$ | 1 | 2 | 3 |
| $H_A$ | $A_{\vdash}$ | 1 | 1 | 4 |
| $H_B$ | $B_{\vdash}$ | 1 | 1 | 4 |
| $I_{AB}$ | $A_{\vdash}$, $B_{\vdash}$ | 2 | 1 | 2 |
| $J_A$ | $A_{\vdash}$ | 1 | 1 | 4 |
| $K_A$ | $A_{\vdash}$ | 1 | 1 | 4 |
| $K_B$ | $B_{\vdash}$ | 1 | 1 | 4 |
| $L_A$ | $A_{\vdash}$ | 1 | 1 | 4 |
| $L_{BA}$ | $B_{\vdash}$, $A_{\vdash}$ | 2 | 1 | 2 |
| $M_A$ | $A_{\vdash}$ | 1 | 2 | 3 |
| $M_B$ | $B_{\vdash}$ | 1 | 1 | 4 |
| $N_A$ | $A_{\vdash}$ | 1 | 2 | 3 |
| $N_B$ | $B_{\vdash}$ | 1 | 1 | 4 |
| $O_A$ | $A_{\vdash}$ | 1 | 2 | 3 |
| $O_{BA}$ | $B_{\vdash}$, $A_{\vdash}$ | 2 | 1 | 2 |

Table 4.6: Displays the trigger case for each pattern type, including the trigger component actions, the number of ordering choices left after triggering the pattern, and the resulting inference power.

| Trigger Component Size | No. Remaining Choices | Inference Power |
|---|---|---|
| 2 | 2 | 1 |
| 2 | 1 | 2 |
| 1 | 2 | 3 |
| 1 | 1 | 4 |

Table 4.7: Determining Power of each pattern, given each distinct trigger Component, and the remaining number of snap action ordering choices.

Figure 4.2 shows how each trigger case for each pattern type, is ranked in relation to each other. This analysis shows us that there are four levels of inferential power. The key attribute values, 'Trigger Component size' and 'no. Endpoint Orderings', that contribute towards this ranking are mapped against each of the pattern trigger cases for each pattern individually in

Table 4.6.

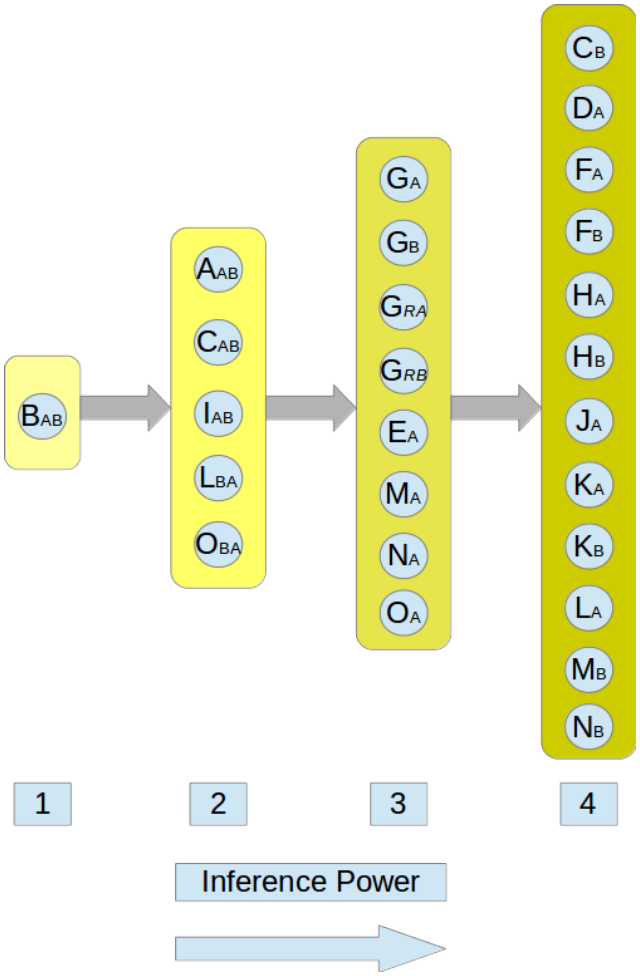



Figure 4.2: Increase in Power of Inference for each distinct trigger case of each pattern type, according to information in Table 4.6.

## 4.5 Pattern Abstraction Safety

In certain circumstances it is possible to compile a pair of actions in a pattern into one action, where the compiled action can still execute concurrently with other external actions where needed. In this section we propose the notion of *Pattern Abstraction Safety* (PAS), and specify when it is safe to compile a pair of durative actions in a pattern structure into a single durative action. The reason for this is so that the pattern of actions can be applied in fewer state transitions, resulting in fewer state generations. The fact that there is required concurrency between the action pair, means that it is guaranteed when one action is selected, that the other is need in the plan as well. Using compilation, the preconditions and effects of the two actions can be combined, the idea being that all of the preconditions become start preconditions and all of the effects become end effects. There are two main benefits that a compilation approach provides which are:

1. The planner does not need to produce individual states for both of the actions in the pattern structure, hence reducing the number of states generated.
2. The planner does not need to perform search in order to determine how the actions in the pattern should be interleaved to be correctly added to the plan because the compiled action is pattern abstract safe.

The second point is a benefit that is achieved with POPI using our infer and search approach. Using compilation, it is a benefit gained by other approaches that do not do the temporal inference that POPI does.

Even though it is possible for some patterns to be compiled so there is only one action, this cannot be done for all pattern types. Some of the pattern types include an actions where effects are deleted and then re-achieved. In this type of situation any action external to the pattern that needs as a precondition a fact which is achieved and then deleted within the pattern, can no longer be applied concurrently with the new compiled action that takes the place of the pair of actions in the pattern. It may also be that the external action may never be applicable even sequentially. For this reason we must define the condition for a pattern to be Pattern Abstraction Safe and determine which pattern types meet this criteria and which do not.

The condition for classifying a pattern as PAS is that there must be no action outside of the pattern being compiled, that needs as a precondition, a fact deleted by an action inside the pattern. This condition for classifying a pattern type as PAS means that all the pattern types described in Chapter 3 that have delete effects are not PAS.

**Definition 4.8** (Pattern Abstraction Safe)**.** Suppose that $a$ and $b$ are two durative actions that are in a pattern of required concurrency and suppose that the intermediate states visited during the execution of these actions through one of the valid paths is s1, s2, s3, s4. Then the pair of actions is considered Pattern Abstract Safe if there is no action $c$ whose preconditions are in one of the states s1, s2, s3, but not satisfied in the state s4.

**Algorithm 3:** compilePatternActions

```
   Data: ActionPair (a, b), lists abPreconditions, abEffects
   Result: Action ab
 1 foreach action act in ActionPair do
 2     foreach precondition x in act do
 3         abPreconditions.add(x)
 4     foreach effect y in act do
 5         if y is negative effect then
 6             return (a, b) notPatternAbstractSafe
 7         else
 8             abEffects.add(y)
 9 action ab
10 ab.startPreconditions = abPreconditions
11 ab.endEffects = abEffects
12 return ab
```

### 4.5.1 Compiling Patterns

Having defined the conditions for a pattern of actions to be PAS, we now go through the pattern types dealt with in this thesis, illustrated in Chapter 3, and show which patterns are PAS and which are not according to those conditions. The condition we propose for a pattern to be Pattern Abstract Safe is simple but robust, where the predicate structure of a pattern type has negative effects, it is deemed to not be safe to compile. We do not deal with negative preconditions in the scope of our work and so this is not a consideration during the PAS analysis. Algorithm 3 shows the process for checking if a pattern of actions is PAS and adding the preconditions and effects of both actions to a new compiled action if it PAS. The pattern types that do not have negative effects and so are PAS include: D, E, F, G, $G_{Reflexive}$, J. The pattern types that have negative effects and are not PAS are: A, B, C, H, I, K, L, M, N, and O. We should note that a pair of actions in a pattern of required concurrency can have other preconditions and effects, that are not part of the predicate structure of the pattern. Therefore it is possible that actions of the pattern types listed above as PAS, can still be unsafe to compile if they have other negative effects that are not part of the pattern structure. The pattern types listed as not being PAS, all have one or more negative effects making up the predicate structure of the pattern itself, meaning actions of these patterns types are never Pattern Abstract Safe, in any instantiation.

### 4.5.2 PAS Example

In order to illustrate how our propose method for Pattern Abstraction works, let us consider pattern D as an example. We can see in Figure 4.3 that Pattern D contains no negative effects in either of its actions. If we input the pattern D action pair into Algorithm 3, we get the compiled action $AB$ as output presented in Figure 4.4.

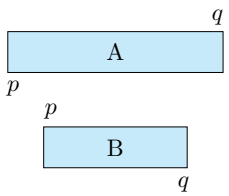



Figure 4.3: Actions in Pattern D structure.

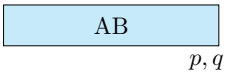



Figure 4.4: Actions in Pattern D structure compiled into one action.

Figure 4.5 shows the state space for applying $A$ and $B$ in their original action format. The letters in superscript are facts needed as preconditions for those actions they are next to and the letters in subscript are the facts achieved as effects by those actions.

Figure 4.6 shows compiled action $AB$ being applied, Any actions other than $A$ and $B$ that require facts $p$ or $q$ as a precondition can use $AB$ to satisfy these preconditions and can be applied at s2.

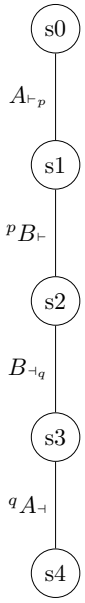



Figure 4.5: Path in search space for applying a pair of actions $A$ and $B$ of pattern type D.

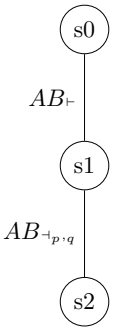



Figure 4.6: Path in search space navigated using compiled action $ab$ in pattern D structure.

### 4.5.3 Non-PAS Example

Pattern A is an example of a pattern structure that cannot be compiled according to the criteria specified for doing Pattern Abstraction. For convenience, we again illustrate pattern A in Figure 4.7. We see that this structure cannot be compiled since the end of A has a negative effect and deletes fact $p$. If we were to attempt to compile this structure, we would effectively end up with an action $AB$ with no preconditions or effects, since the only fact achieved is also deleted and $p$ as an invariant of $B$ would be hidden as a result of the compilation and would not be visible in the single compiled version of the action.

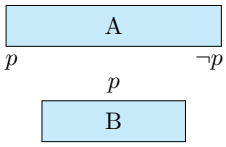



Figure 4.7: Example of Pattern A structure which is not Pattern Abstract Safe.

## 4.6 Summary

In this chapter we have provided a set of theorems that show all the forms of pair-wise required concurrency that are possible in a general form, according to sets of concurrent application sequences for the temporal propositional case. We have shown how the pattern types in Chapter 3 are each an instantiation of the generalised pattern sets presented in Section 4.3 and how they correspond. Furthermore, we have provided a method to classify the inferential power of each of the patterns according to each specific trigger case. Finally, we have presented a method for compiling certain pattern type structures which can prove beneficial during state space exploration and also shown which pattern types cannot be compiled, strengthening the motivation and usefulness of our approach in applying the actions in those pattern types structures using our inference techniques. In Chapter 5 we will provide a method for measuring the benefit of the inferences from each pattern type using information theory. This measurement will be of the information gain of using the approach of POPI over its baseline planner POPF and Breadth-First Search.

# Chapter 5

# Information Gain from Patterns

In this chapter, we present an information theoretic analysis of each pattern type described in Chapter 3. We will discuss the information content of each pattern type, per trigger case and the information gained using inference versus not using inference and using search instead. In Chapter 4 we saw how the inferential power of each pattern, according to each of its trigger cases varies and presented a system to categorise the inferential power of each pattern case on one of four levels. This analysis will provide a perspective of the gain that may be achieved from the patterns in the reduction of choices that must be made during planning and generation of the state space.

We use Shannon's theory of communication (Shannon [1948]) and his formula for calculating entropy to measure the information gained by POPI when adding a pair of actions in a pattern using its combined search and infer process, instead of using an alternative state exploration strategy. The reason for using information theory is because we are interested in analysing what information POPI gains and knows it can exploit after triggering a pattern, enabling the use of inference, compared with other approaches. Our approach for this analysis is to measure the information gain of POPI over our baseline planner POPF, in order to determine the theoretical increase in the amount of extra inference POPI can do when pattern trigger actions become applicable in a state during planning. We will explain the context of our problem scenario, the goal of which will be to add the two actions for each pattern trigger case to a plan, and compare the choices that need to be made by each strategy being compared. The difference in information content being measured is between when search is used to generate a state, and when inference is used to generate a state, when it is pattern actions being applied. We will discuss the planner POPI in its theoretical context and the amount of gain it attains in information about how the rest of the actions in a pattern must unfold. We also measure POPI's information gain over Breadth-first search (BFS) which blindly explores the state space  without heuristic information. The reason for this is that we also to provide a view of the maximum benefit that POPI brings as well as its improvements

over the baseline planner POPF.

## 5.1 Measuring Information Gain

Required concurrency is known to exist following the commitment to the trigger action component of the pattern, which is either the start of $A$ or $B$ or both. Inference is certainty in knowing how actions must be applied having discovered that required concurrency exists. We propose which state transition choice points can in principle be avoided by POPI. The information gain in each pattern trigger case is calculated by counting the number of state transitions that POPI must make to apply the actions in the pattern structure, as opposed to the search strategy being compared with. As some patterns have two trigger cases for how inference is enabled, we calculate the information gain of POPI against the other strategies for each pattern trigger case individually, as they can vary.

The reason for counting state transitions is because we want to consider BFS for comparison, where state evaluation is irrelevant. We also want to consider our baseline planner POPF, which evaluates some states but not necessarily all states, depending on whether it has proven a state inconsistency via temporal reasoning or through detecting a violation of active invariants. The counting of state transitions allows us to abstract away from the state operation, which is the generation of the state and perhaps its evaluation, depending on the strategy being used. This gives us a common method for comparison and allows us to focus on measuring the gain by eliminating choice in the selection of the actions used to perform the state transitions. POPI can in principle short circuit the generation and evaluation of some states by being able to apply the combined effects of actions inside a pattern in a single state transition. All of the effects of each pattern of actions, according to one of it valid application sequence(s) can in theory be applied in one operation and generating a single successor state.

Practically in its implementation, POPI still generates and evaluates intermediary states for book-keeping purposes, but does not use the heuristic evaluations of intermediary pattern states. However, these are implementation details of how state operations occur and are not important when assessing the theoretical reduction in the number of state transitions that need to be chosen, using the combined inference and search strategy of POPI compared with other search strategies. When required concurrency is detected by POPI following the selection of the trigger action(s), the sequence of state transitions in the path to the completion of the pattern are inferred. Certainty in knowing a valid sequence of action applications that must happen as the next state transitions, following the application of the trigger action(s), is the gain in information being measured.

To calculate the information gain, we must define which state transitions are counted. For BFS and POPF all state transitions are counted, since a choice must be made at each state about which action to generate the next state with. For POPI only the state transitions from the application of the trigger component action(s) are counted. This is because the state transitions resulting from the remaining actions in the pattern are generated from the inferred sequence of actions that constitute the reduction in search choices, achieved by POPI triggering a pattern. POPI does not need to make a search choice at the intermediary pattern states generated via the inferred sequence of actions. All of the inferred actions are treated as an extension of the initial choice to select the trigger action(s).

### 5.1.1 Problem Context

In order to perform a consistent calculation of the information gain of POPI in each pattern case, we need a consistent problem situation in which to depict and compare the state space expansion processes for the problem solving systems that are being compared; these being Breadth-first search, POPF and POPI. For this reason the problem scenario within which the pair of actions in each pattern structure will be added to the plan is set up as follows. The only actions that exist are $A$ and $B$, each of which needs to be applied to reach the goal and nothing else. This is because we want to focus the analysis of the information gain with regards to the state space expanded using the actions in the pattern for the pure atomic case of the problem. Allowing other actions in the state space in addition to the pattern actions does not help us to calculate the information gain produced by the patterns and complicates the calculations and analysis unnecessarily. Actions $A$ and $B$ are also both one-shot actions and no other groundings of the action exist.

Although this problem scenario is simple and highly restrictive, this is so that we can determine the information gain achieved without interference from other external factors. This will allow us to attain a precise set of measurements that can be compared based on a common set of ground rules. We decided to make the problem such that both $A$ and $B$ both achieve a goal fact, $ga$ and $gb$ respectively meaning both need to be applied. This is because in the case of some patterns, either $A$ or $B$ can be applied on its own and we are interested in what happens when required concurrency exists; these patterns require the starts of A and B to be applied to enable inference. As there are two trigger cases for some patterns, where this is the case, the paths navigated by POPF and POPI in both trigger cases of a pattern will be included in the state space diagrams for that pattern. As for BFS, the strategy itself means that it expands states using all applicable actions at each level of the state space from left to right, before progressing to the next level. Therefore we do not display a second goal state on the right side of the state space for BFS, if one is already found on the left since it would be part of the same path navigated from the start. The POPF and POPI state spaces

are illustrated to display an 'either or' situation; the two paths to the goal represent mutually exclusive paths according to how the pattern triggers.

The rules for our state space expansions and information gain analysis are as follows:

1. The facts that exist as preconditions and effects for the actions in each pattern are those in its example pattern structure, as illustrated in Chapter 3. The only other facts that exist are those making the pattern actions one shot and each action so it achieves one of the two goal facts, *da* and *db*. These additional facts are displayed in Figure 5.1.
2. One instance of each action in the pattern may be used to achieve the goal (one shot actions).
3. In all strategies, we assume regression of invariants through action states rule is used, meaning that invariants on actions are also treated as start preconditions. We utilise this rule for our analysis in the same way that the POPF planner does as described in subsection 2.8.11 in Chapter 2.
4. Both actions must be applied, each action achieves one of two goal facts (goal is for pattern to be completed).
5. Compression safety is not utilised during state expansion.
6. Assume action durations may be of whatever length required to support any valid sequence of action application, given the preconditions of action endpoints.
7. Only state transitions are counted, abstracting away from whether this results in only state generations or state evaluations as well.
8. We look at the worst case and assume that where the wrong choice may be made by a strategy, that this happens. In the case of POPF and POPI which use heuristic guidance, where there is an arbitrary choice and the wrong action can be selected, it is.
9. We assume state memoization is used during state expansion by all strategies, such that we do not consider state spaces where the same two actions are repeatedly applied, producing the same states an arbitrary number of times.

The gain in information content using POPI's approach over Breadth-first search (BFS) and POPF is calculated as:

$$\text{Information Gain} = -log_2(P_i) \ ,$$

where $P_i$ is the number of state transitions generated via search by POPI divided by the

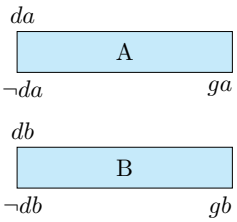



Figure 5.1: One shot action pair, each achieving one goal condition.

number of state transitions used by the search strategy being compared with (BFS or POPF), in order to successfully apply a pair of actions $A$ and $B$ in pattern of required concurrency via a valid sequence of application.

## 5.2 Measuring Information Gain from Pattern Cases

In this section we will look at the number of bits of information gained via the inference of each pattern trigger case and discuss each of them in turn. The search space diagrams for each of the three strategies are presented as sub-figures for each pattern, displayed in Figures 5.3– 5.18 for pattern types A, B, C, D, E, F, G, $G_{Reflexive}$, H, I, J, K, L, M, N, O respectively. Figure 5.2 shows the full state space for all the combinations in which a single instance of actions $A$ and $B$ may be applied, not taking into account any particular pattern. The state space expanded by each strategy is a subset of this state space, for all pattern types.

### 5.2.1 State Space Diagrams

For the Breadth-First Search and POPF sub-figures, all state transitions are always selected as a choice via search. This includes end actions held in the event queue by POPF, which although it knows must be applied, it still makes a choice to use as the transitions from particular states. In order to understand the state space diagrams, we colour code the all the state space diagrams as follows. The initial state will be highlighted in yellow; this is where at least one of the start actions in a pattern becomes applicable. White filled nodes represent states where a choice is made to select an action. States highlighted in red mean that those states have been generated and evaluated but found to be dead-ends.

For the sub-figures displaying POPI's state space, in addition to the colour coding mentioned so far, orange is also used for states which were generated from a search chosen action the same as for BFS and POPF, but all make up a trigger component for a pattern. Finally, for POPI, the green represents the states produced from inferred action state transitions. The path from the orange state, or the second orange state if there are two, to the last green state

is the inferred sequence of actions and states and these transitions are not counted for POPI. A key point to understand is that once a pattern has been triggered, the remaining sequence is a consequence of selecting the trigger action transition(s) for that pattern. The inferred sequence is viewed as an extension of adding the trigger action(s) to the plan.

We now present the state space navigated by each strategy for each pattern type. Along with the illustrations of these state spaces we will discuss the different strategies with regards to their navigation of the state space. We can observe from the outset that all the states generated by POPF and POPI are the same, which is logical since there are only two one-shot actions in the problem context for each of these state space examples. The difference is that POPF makes a choice at each state when going to apply an action, whereas POPI applies the inferred actions without considering alternative applicable actions.

In the interest of conciseness, for the patterns where there are two trigger cases, we have included the paths navigated for the two trigger cases into single diagrams, but still one per pattern type. This only applies for POPF and POPI, since the traversal of the search tree by BFS is the same regardless of the trigger case. When viewing these diagrams and there are two paths to the goal state, one starting with action $A_{\vdash}$ and the other with $B_{\vdash}$, the calculation for the information gain in each case is performed using only the number of state transitions in the path for the trigger case being described.

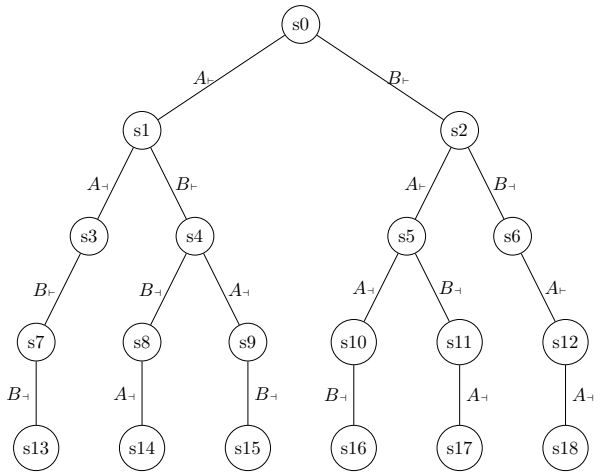
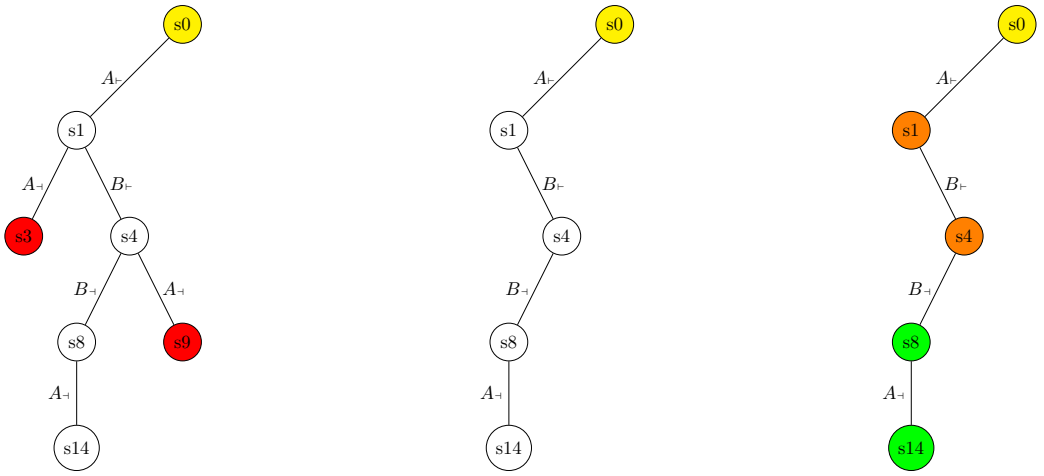



Figure 5.2: Possible State Space for a pair of actions in a pattern structure.

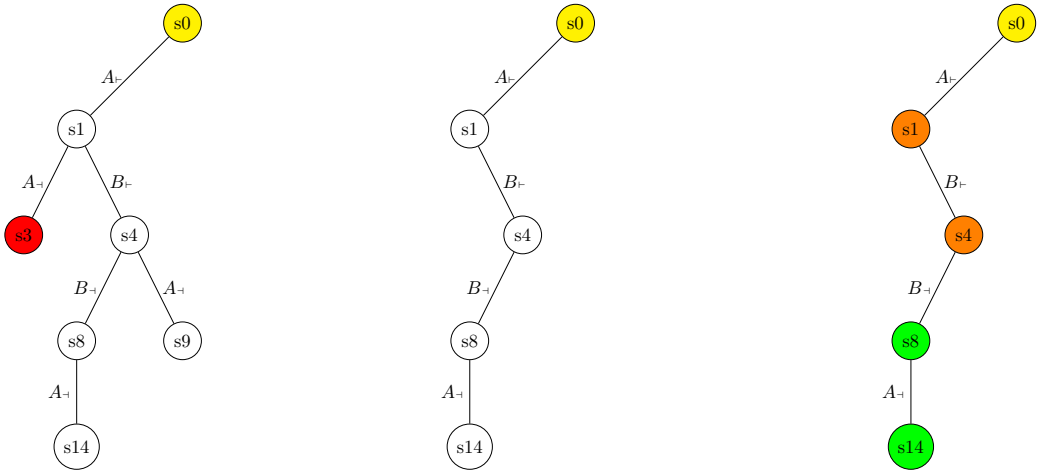
In Figure 5.3a we observe that BFS generates state s3 using the end of action $A$ because it was applicable, but is a dead-end, given rules 2 and 4. We can also see that BFS generates state s9, since the end of action $A$ is applicable. However, this is a dead-end state since it violates the invariant of action $B$ meaning the goal can no longer be achieved. POPF using its heuristic can determine that $B$ is needed to achieve the goal as does POPI, causing them both apply the start of $B$ after the start of $A$. Figure 5.3b shows that POPF knows that both of the actions started must also be finished and there are no other actions needed to achieve the goal. POPF's heuristic guides it to apply the start of $A$ followed by the start of B rather than simply applying the end of $A$ immediately, which would result in a dead-end. The pattern A structure informs POPI of the same application ordering given the ordering constraints of pattern A, and POPI will not make any search choices at states s4 and s8. The diagram for POPI's application of the actions in pattern A is shown in Figure 5.3c. The lay-out of the sub-figures for the remaining state space diagrams is the same, with Breadth-first search, POPF and POPI as sub-figures $(a)$, $(b)$ and $(c)$ respectively.




(a) Breadth-First Search. (b) POPF. (c) POPI.

Figure 5.3: States expanded to apply actions in a pattern A structure.

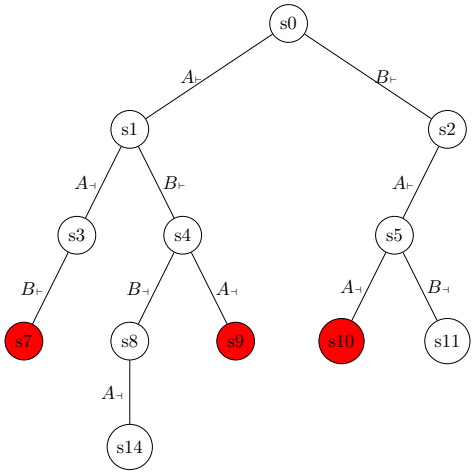
Figure 5.4 presents the state spaces navigated using a pair of actions in a pattern B structure. Although the states generated by BFS are the same as those for its situation in pattern A. The difference here is that state s9 is not a dead-end. The reason that the search is completed with A⊣ after B⊣, is because BFS expands each level from left to right, and assuming the same arbitrary ordering of actions, s14 is again reached as the goal. State s15 could exist from applying B⊣ after A⊣ produces s9 is alternative goal state, but s14 is reached first and so BFS terminates. POPF and POPI expand the state space in the same manner as for pattern A, however B⊣ and A⊣ could be applied in either order and the constraints of the action pair allow for this.



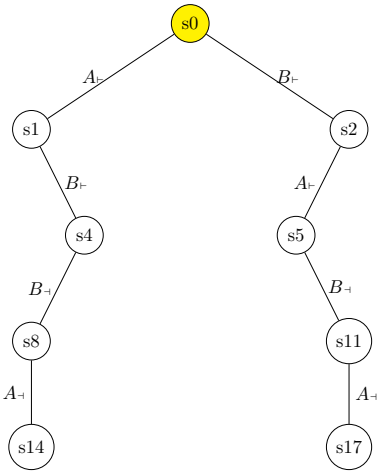

(a) Breadth-First Search. (b) POPF. (c) POPI.

Figure 5.4: States expanded to apply actions in a pattern B structure.

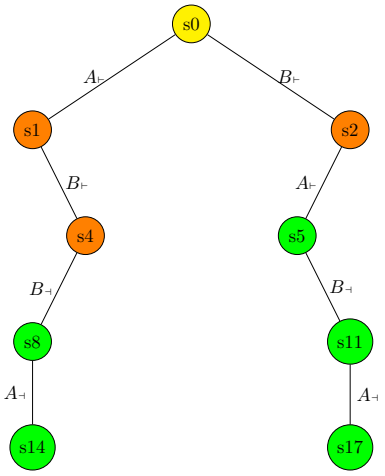
The state space for pattern C shown in Figure 5.5 has two trigger cases. For BFS we see that using its uninformed search it reaches a dead-end on three occasions in states s7, s9 and s10. The first goal state reached is s14, after 11 state transitions have been used. The information gain of POPI over BFS here is therefore higher than in the case of patterns A and B. The information gain of POPI over POPF is higher for trigger case $C_B$ than it is for the $C_{AB}$ trigger case, since there is only one trigger action for the $C_B$ case and the rest of actions are inferred as shown in Figure 5.5c.




(a) Breadth-First Search.



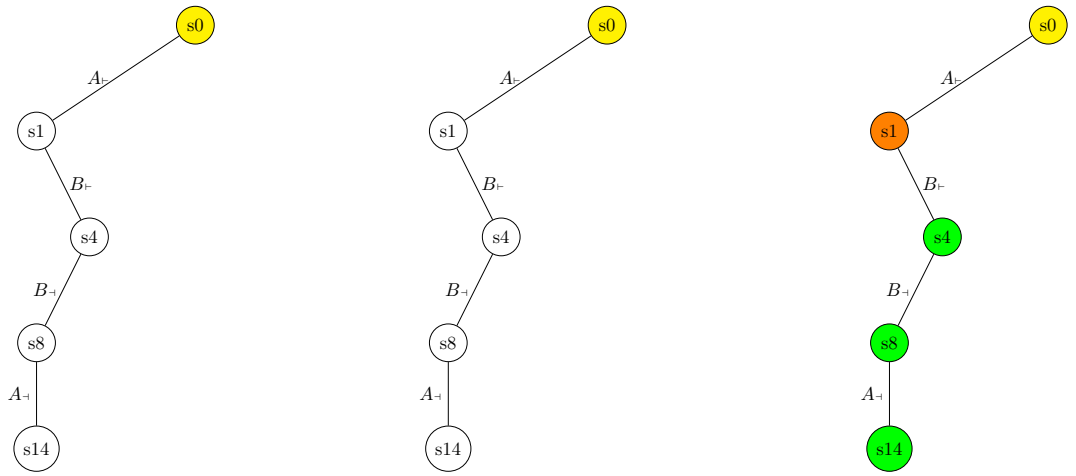

(b) POPF.




(c) POPI.

Figure 5.5: States expanded to apply actions in a pattern C structure.

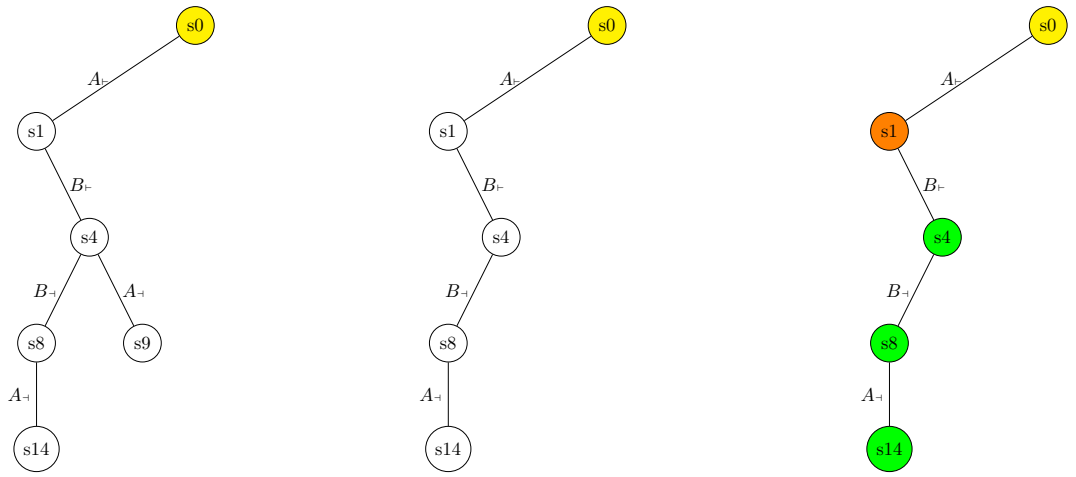
The state expansions for pattern D shown in Figure 5.6 presents the first time where the navigation by all the strategies is the same. The reason for this is that pattern D has only one order in which its snap actions can be applied, which is A⊢, B⊢, B⊣, A⊣. This is because of the precedence constraints between actions $A$ and $B$. Even though BFS does a blind state space exploration, since there are only four snap actions and one order in which they become applicable, its navigation of the state space is the same as POPF and POPI. However, POPI benefits significantly in information gain from knowing that required concurrency exists between $A$ and $B$ and infers the addition of $B$ to the plan and applies actions B⊢, B⊣, A⊣ via inference instead of search.




(a) Breadth-First Search. (b) POPF. (c) POPI.

Figure 5.6: States expanded to apply actions in a pattern D structure.

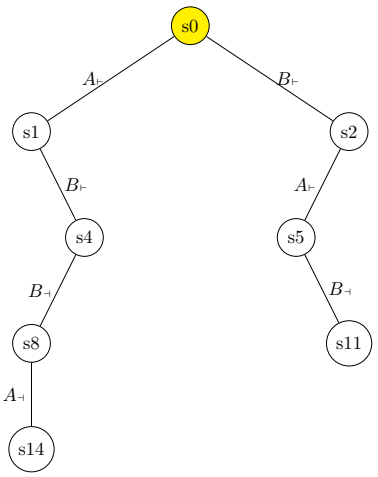
The situation for pattern E shown in Figure 5.7 is similar to that of pattern D, the difference is that for BFS, state s9 is explored before the goal is reached in state s14. Pattern E is less restrictive and allows A⊣ and B⊣ to be applied in either order. Since A⊣ is applicable at state s4, BFS generates state s9 before it generates s14 reaching the goal, as seen in Sub-figure 5.7a. This means that the number of state transitions used by BFS for pattern E is one more than in pattern D, hence the information gain of POPI over BFS will be higher than for pattern D case, but is the same for POPI over POPF.



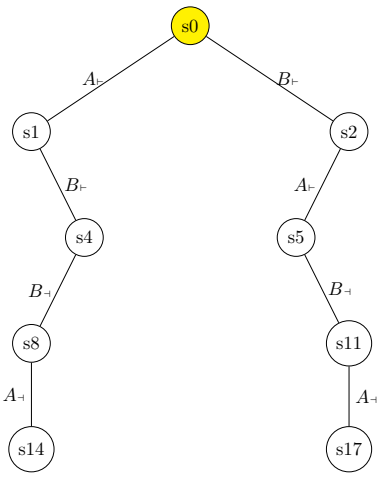

(a) Breadth-First Search. (b) POPF. (c) POPI.

Figure 5.7: States expanded to apply actions in a pattern E structure.

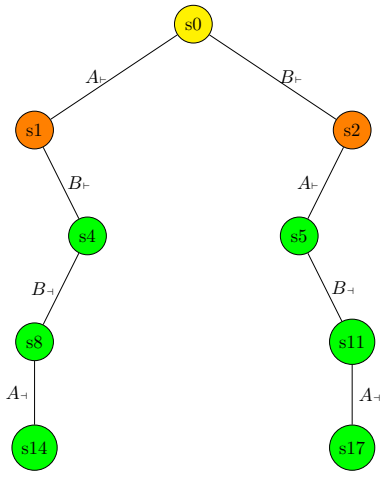
For pattern F shown in Figure 5.8, there are two trigger cases. BFS does not reach a dead-end and arrives at the goal state s14. In both trigger cases, POPI triggers for inference after only one start action is added to the plan and the other three remaining actions are inferred. Its gain in information over POPF is the same in both trigger cases, $F_A$ and $F_B$.




(a) Breadth-First Search.




(b) POPF.

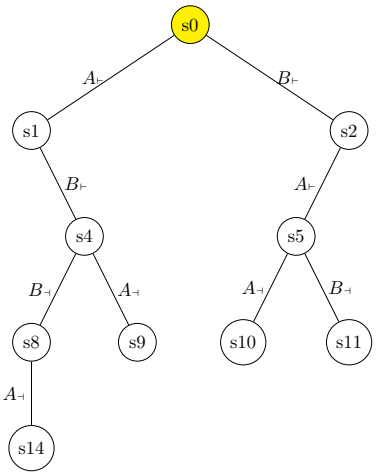



(c) POPI.

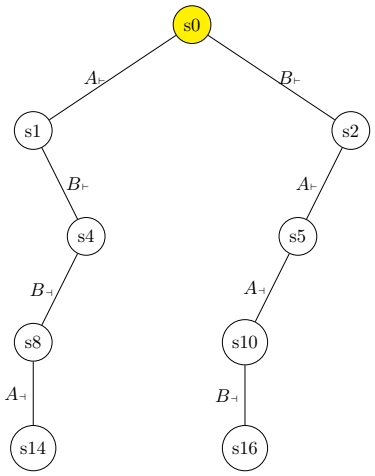
Figure 5.8: States expanded to apply actions in a pattern F structure.

Pattern G is a situation where BFS, shown in Sub-figure 5.9a, explores many states before reaching a goal state. This is since the constraints of the actions allow for more flexibility in terms of the concurrent orderings compared to the other pattern types. We can see that BFS generates nine states to reach the goal. This is the same regardless of the trigger case, whereas the number of states generated can vary between trigger cases for POPI, between the two trigger cases of a pattern type, as is the case for pattern type C presented in Sub-figure 5.5c.

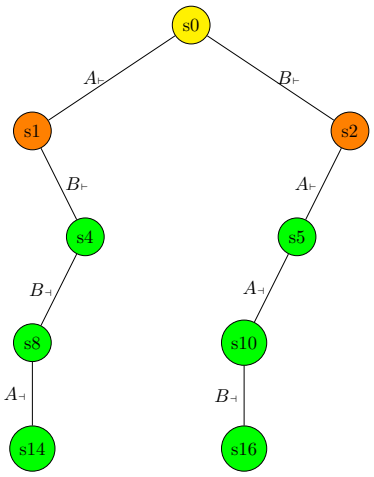
Although there are four concurrent orderings in which actions in a pattern G structure can be applied, POPI knows that all of these are valid, and orders the snap actions in the pattern, into a valid sequence of application that requires the least amount of reordering. This is why we do not see more states in the state space for POPI in Sub-figure 5.9c. POPF sees the same states generated shown in Sub-figure 5.9b, however following the initial state all of these states are still generated via EHC, where the actions were chosen to generate its states.




(a) Breadth-First Search.



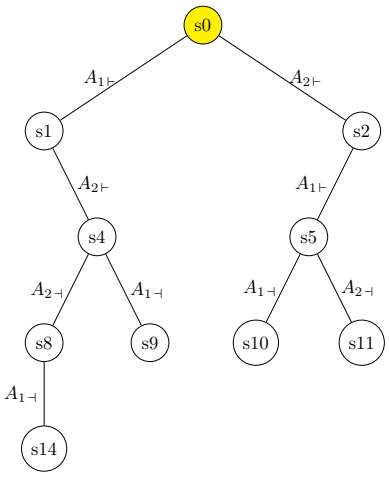

(b) POPF.



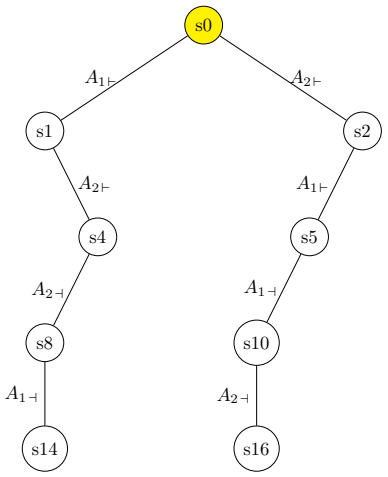

(c) POPI.

Figure 5.9: States expanded to apply actions in a pattern G structure.

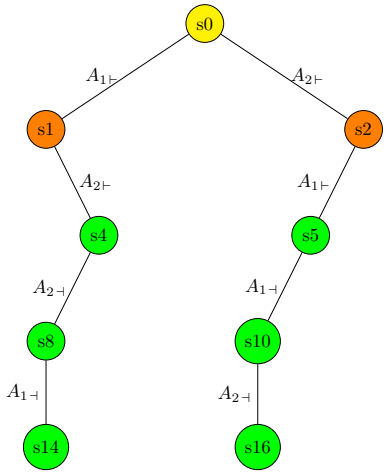
Pattern $G_{Reflexive}$ shown in Figure 5.10 is the same as pattern G, except the required concurrency is between two instances of the same operator; the same analysis of the pattern G state spaces applies.




(a) Breadth-First Search.



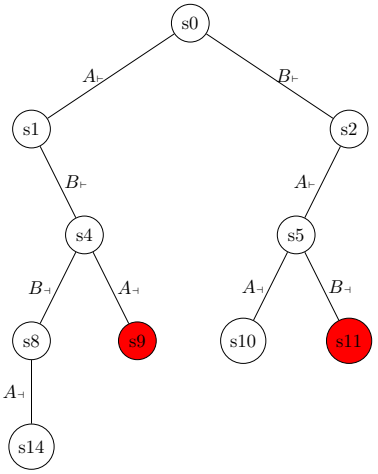

(b) POPF.



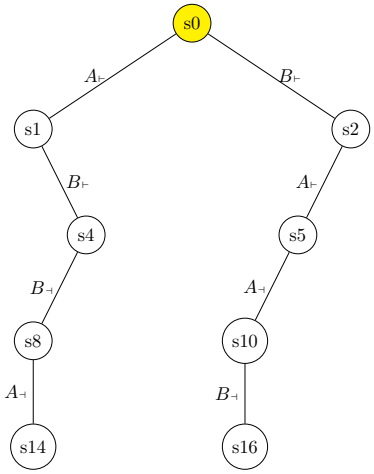

(c) POPI.

Figure 5.10: States expanded to apply actions in a pattern $G_{Reflexive}$ structure.

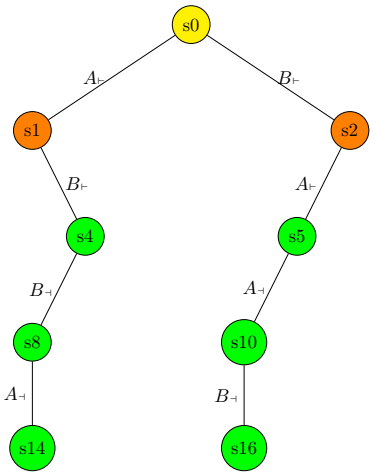
In the pattern H scenario shown in Figure 5.11, we see that BFS reaches a dead-end twice before getting to a goal state. In this pattern structure, there are two valid action application sequences, one per trigger case. POPI has a high information gain over BFS in both trigger cases, because in both cases BFS iterates through nine state transitions to reach the goal and POPI applies four, but only the trigger action is applied via search. The triggering of the pattern informs POPI with certainty in both of the trigger cases, that the remaining three actions can be applied in a particular order, according to the precedence constraints enforced by the pattern. The path navigated by POPF is the same as POPI, but again a search choice is made at each state in selecting the action for the state transition.




(a) Breadth-First Search.



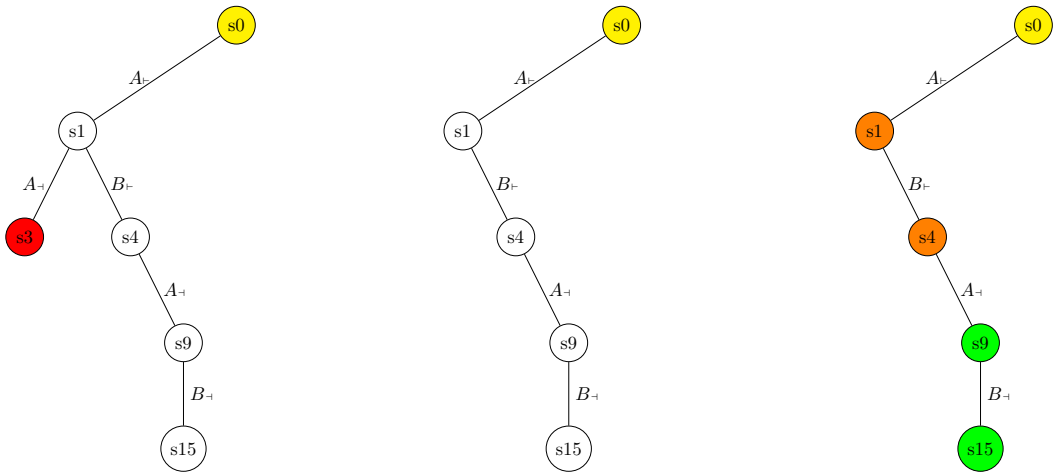

(b) POPF.




(c) POPI.

Figure 5.11: States expanded to apply actions in a pattern H structure.

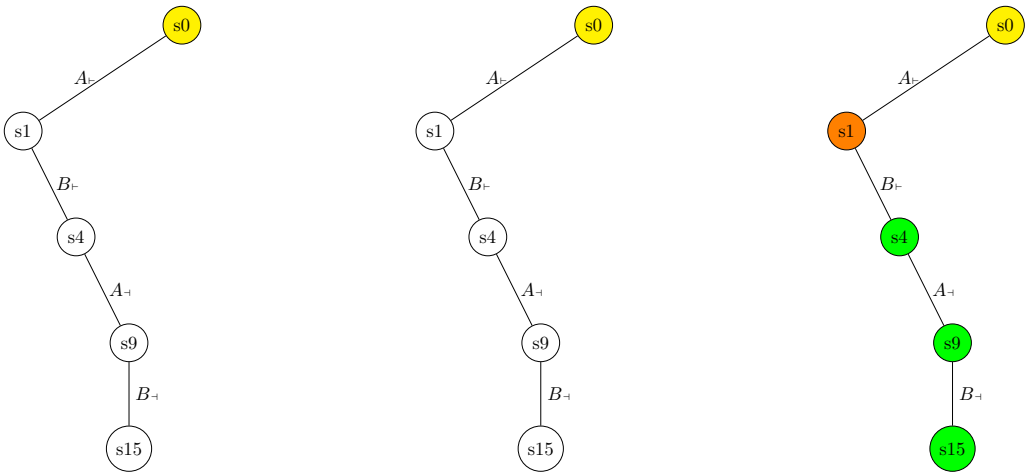
The navigation of the state space with a pattern I structure for BFS in Figure 5.12a, shows that one dead-end is reached before navigating down the path to the solution. Pattern I portrays a one way consumer and producer relationship between a pair of actions $A$ and $B$. This means that there are two trigger actions and two states generated via search after the initial state. This added to the fact that BFS only explores one other state in its state space (the dead-end) compared to POPI, means that the information gain of POPI over BFS is lower than in other more powerful patterns such as pattern H seen above. The information gain of POPI over POPF is even smaller, since again POPI only applies the state transitions resulting in states s9 and s15 via inference.




(a) Breadth-First Search. (b) POPF. (c) POPI.

Figure 5.12: States expanded to apply actions in a pattern I structure.

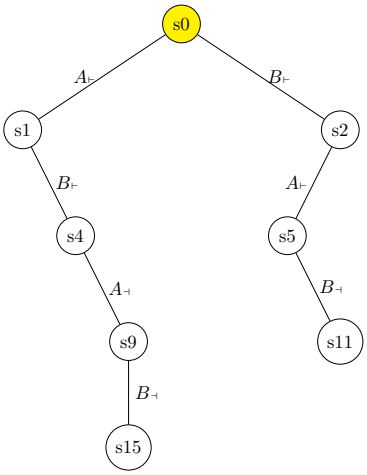
In the pattern J state spaces in Figure 5.13, we see the exact same situations as in pattern D, except that the order of the snap actions is different. The number of states generated via search and inference is the same amongst all of the strategies as it is for pattern D and so is amount of information gain of POPI over BFS and POPF. Since actions $B_{\dashv}$ and $A_{\dashv}$ are applied the other way round than in pattern D, the paths navigated in the search space compared to pattern D are different. After state s4, states s9 and s15 are generated, instead of states s8 and s14 as is the case for pattern D. However, this has no effect on the information gain calculations and is simply state labelling to distinguish clearly the different action application orderings.



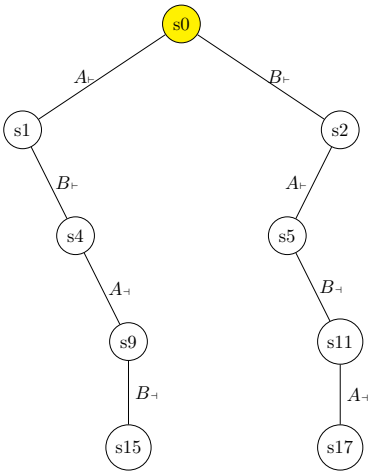

(a) Breadth-First Search. (b) POPF. (c) POPI.

Figure 5.13: States expanded to apply actions in a pattern J structure.

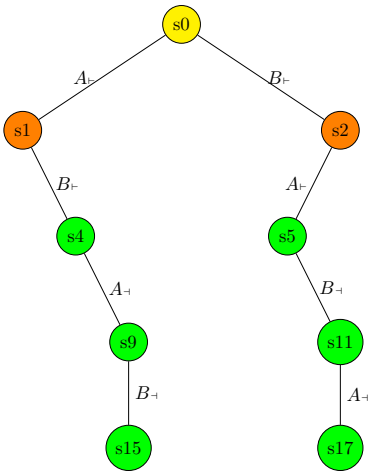
Figure 5.14 presents the state spaces for pattern K. The information gain of POPI over BFS is the same for both of its trigger cases, $K_A$ and $K_B$. This is also true for POPI over POPF. The amount of information gain is also identical to pattern F which can be seen in Table 5.1. The states generated and the set of paths taken to the solution for POPF and POPI are different, since the valid sequences of action application are different; this is what distinguishes them as different patterns. We can observe that between pattern F and K, one sequence of action application is the same which is $B_{\vdash}$, $A_{\vdash}$, $B_{\dashv}$, $A_{\dashv}$ and the other is different. As we observed in the comparison of pattern D and J above, this has no effect on the information gain. The reason for the information gain being the same for POPI over BFS and POPI over POPF between pattern F and K, is because the number of state transitions is the same for each strategy in both of these patterns and so are the number of actions searched for by POPI.




(a) Breadth-First Search.



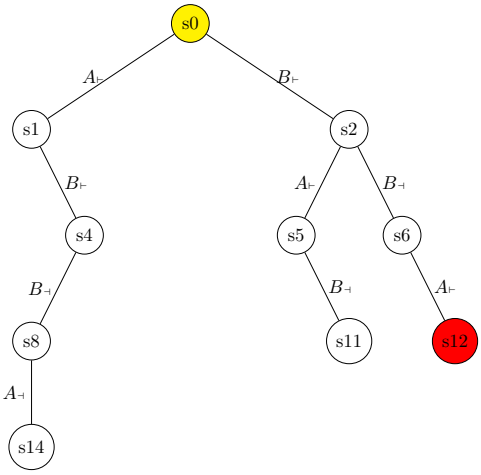

(b) POPF.



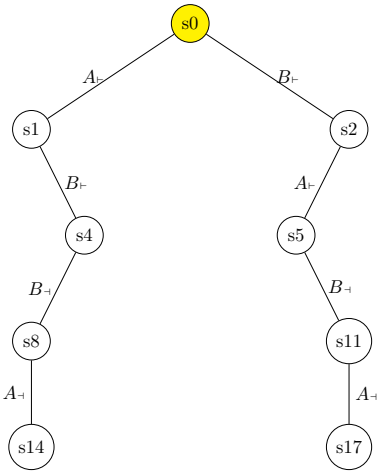

(c) POPI.

Figure 5.14: States expanded to apply actions in a pattern K structure.

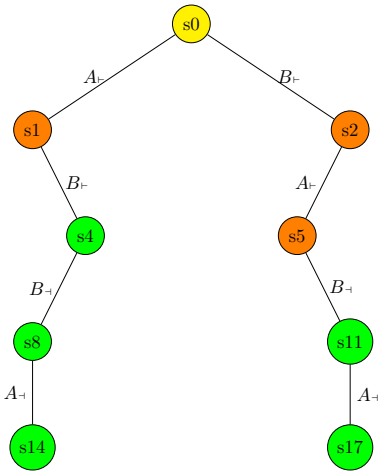
The state spaces for pattern L shown in Figure 5.15 display a situation where the information gain is high for trigger case $L_A$ for POPI over BFS, this is due to their being nine states generated by BFS in order to reach the goal including one dead-end state generated immediately before the goal state is reached. The amount of information gained over BFS in the second trigger case $L_{BA}$ is lower since POPI needs to apply two actions via search before inference is enabled. For POPI over POPF, the situation is the same where case $L_A$ allows more information gain than case $L_{BA}$.




(a) Breadth-First Search.



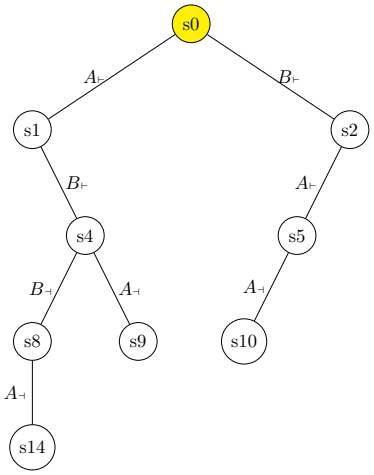

(b) POPF.



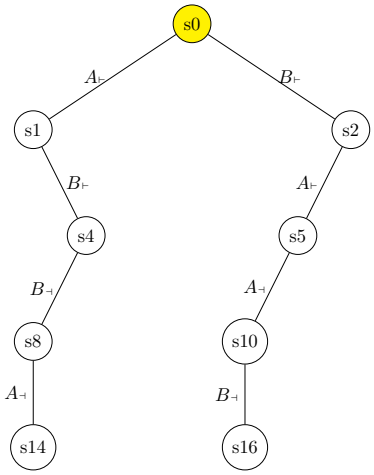

(c) POPI.

Figure 5.15: States expanded to apply actions in a pattern L structure.

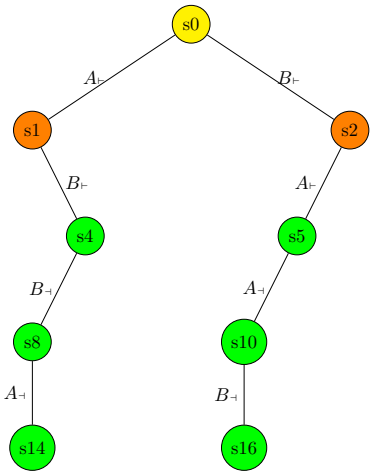
The pattern M states spaces in Figure 5.16 show that POPI triggers for inference after one action added via search in both trigger cases. We see in Sub-figure 5.16a that BFS does not generate any dead-end states in the search space for this pattern. BFS reaches the goal at state s14 since it is the first goal state it generates. State s15 that ends the plan with action B⊣ transitioning from s9 is also a viable alternative for both POPF and POPI. In the case of POPF this would be an arbitrary ordering and for POPI, actions are re-ordered only to satisfy the precedence constraints in the pattern. This would not affect the amount of information gained by POPI over POPF, which is the same in both trigger cases. The full combinatorial state space, for which the search spaces shown in Figures 5.3 to 5.18 are a subset, is presented in Figure 5.2.




(a) Breadth-First Search.



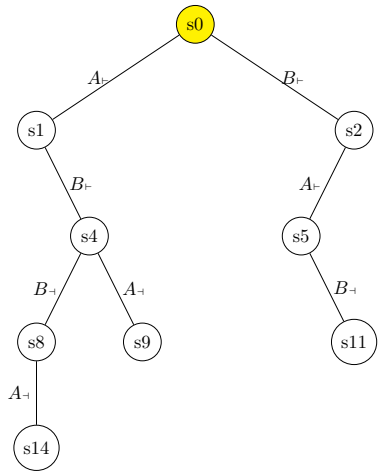

(b) POPF.



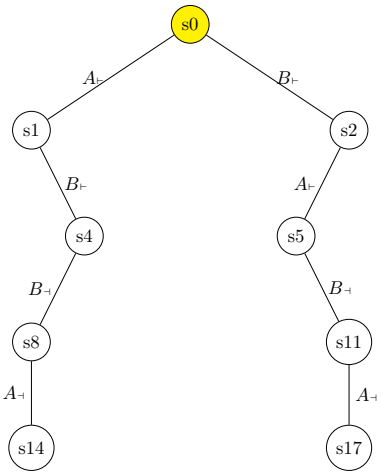

(c) POPI.

Figure 5.16: States expanded to apply actions in a pattern M structure.

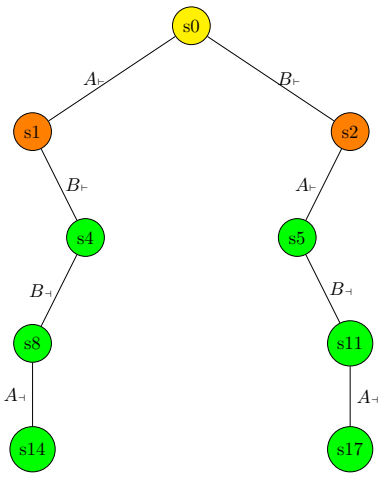
The pattern N state spaces in Figure 5.17 present a set of situations that are the same as their counterparts for the pattern M state spaces, in terms of the number of state transitions used and states generated. This is an observation that we have already seen between other patterns. The exact combination of state transitions used are different resulting in a set of state generations. Since the amount of information gain achieved by POPI over BFS and POPF, depends on the number of state transitions used via search compared with the number used via search by the others to reach the first state achieving the goal, the amount of information gain is the same is all cases for pattern N as it was for pattern M in both trigger cases.




(a) Breadth-First Search.



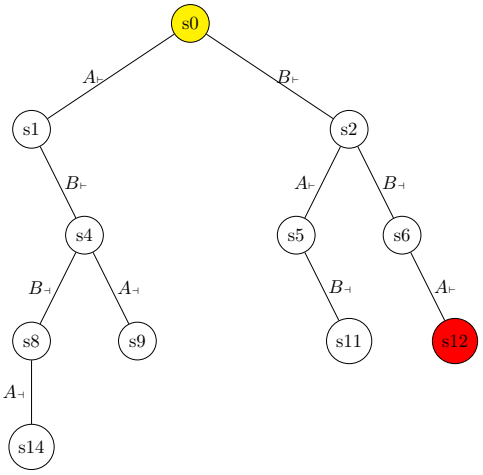

(b) POPF.



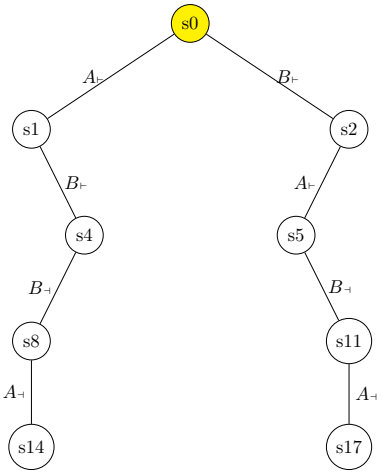

(c) POPI.

Figure 5.17: States expanded to apply actions in a pattern N structure.

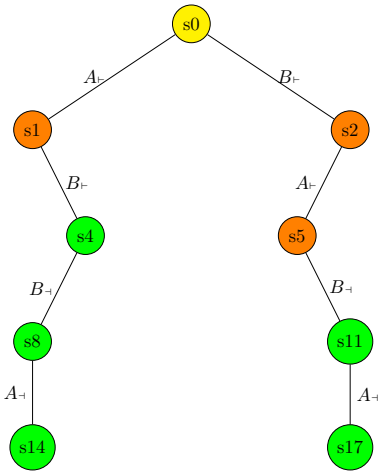
Figure 5.18 displays the state spaces for the pattern O pair of actions. In Figure 5.18a BFS reaches a dead-end at state s12 after having applied three actions because $A_{\dashv}$ is not applicable and the goal cannot be achieved. The information gain for POPI over the other two solving systems is different between the two trigger cases. The information gain achieved in the pattern O state space is the highest for POPI over BFS in trigger case $O_A$, and is the lowest with POPI over POPF in trigger case $O_{BA}$.




(a) Breadth-First Search.




(b) POPF.




(c) POPI.

Figure 5.18: States expanded to apply actions in a pattern O structure.

Table 5.1 displays the information gain of POPI over Breadth-first search and POPF, our baseline planner. The reason for comparing against these two solving systems was, in the case of BFS to give us an idea of POPI's maximum level of inferential power in the best case (worse case performance for BFS), and then a moderate view of POPI's benefit measuring

the extra gain it brings in addition to the heuristic search and inference that POPF already does. We provide the information gain of POPI per trigger case, since as we have already seen from the state space diagrams of some patterns with two trigger cases, that the inference which POPI can do in each trigger case can be different in its significance. Particularly for patterns types like pattern C where trigger case $C_B$ enables the addition of a new action to the plan That is a more powerful inference than trigger case $C_{AB}$ which only allows inferring the ordering constraints between actions $A$ and $B$, which have already been added to the plan via search. It should be noted that all information gain measurements for POPI over another planning strategy (BFS or POPF) refer to the aggressive strategy's gain (POPI-AI) over that strategy. The last row in Table 5.1 shows the amount of information gained by POPI over BFS and POPF when averaged over all of the trigger cases for all pattern types.

| Pattern Trigger Case | POPI/BFS | POPI/POPF |
|---|---|---|
| $A_{AB}$ | 1.585 | 1 |
| $B_{AB}$ | 1.585 | 1 |
| $C_{AB}$ | 2.459 | 1 |
| $C_B$ | 3.459 | 2 |
| $D_A$ | 2 | 2 |
| $E_A$ | 2.322 | 2 |
| $F_A$ | 2.807 | 2 |
| $F_B$ | 2.807 | 2 |
| $G_A$ | 3.167 | 2 |
| $G_B$ | 3.167 | 2 |
| $G_{Reflexive_A 1}$ | 3.167 | 2 |
| $G_{Reflexive_A 2}$ | 3.167 | 2 |
| $H_A$ | 3.167 | 2 |
| $H_B$ | 3.167 | 2 |
| $I_{AB}$ | 1.322 | 1 |
| $J_A$ | 2 | 2 |
| $K_A$ | 2.807 | 2 |
| $K_B$ | 2.807 | 2 |
| $L_A$ | 3.167 | 2 |
| $L_{BA}$ | 2.167 | 1 |
| $M_A$ | 3 | 2 |
| $M_B$ | 3 | 2 |
| $N_A$ | 3 | 2 |
| $N_B$ | 3 | 2 |
| $O_A$ | 3.322 | 2 |
| $O_{BA}$ | 2.322 | 1 |
| Average | 2.69 | 1.769 |

Table 5.1: Information Gain from pattern based inferences by POPI over Breadth-first Search and POPF.

## 5.3 Summary

In this chapter we have provided an information theoretic perspective of the pattern types that we consider in this thesis. We provide a method of measuring the inferential power, in each pattern trigger case in bits of information gained, where each bit of information gained by POPI over another solving strategy, equates to some proportion of search choice(s) that POPI did not have to make to reach the goal. Information gain allows us to measure how much more POPI knows about which actions need to be applied and in what order, compared with the other two state exploration strategies. We have seen that generally the amount of information gained by POPI over BFS is greater than POPI over POPF. This is logical since BFS explores the state space blindly without heuristic guidance. Furthermore, POPI is an extension of POPF and its search component and method for choosing actions via search is the same as POPF, at states where no pattern structures exist.

# Chapter 6

# POPI

## 6.1 Overview

In this chapter we describe the details of the planner POPI, which implements the pattern detection and inferences discussed in Chapter 3. We describe the pattern detection algorithms that have been designed to perform a pre-search analysis of the domain structure, for domains containing durative actions written according to the specifications of PDDL 2.1 (Fox and Long [2003]).

The method for detecting required concurrency works in two core parts. The first part works by analysing the domain structure and extracting relevant information from the condition and effects lists of the action schemas and storing possible candidates in map structures. The second part works by comparing the lists of candidate actions attached to each candidate predicate and matching operator pairs that have the predicate structure of the pattern type. The patterns accommodated by the pattern detection in POPI's implementation are the ones with two distinct operators recorded, therefore $G_{Reflexive}$ is not included.

## 6.2 Implementing POPI

The planner POPI has been built as a modification and extension to POPF. It implements a pattern matching approach for detecting the patterns described in Chapter 3, which are instantiations of the generalised pattern sequence sets they are associated with described in Chapter 4. Each pattern type can be achieved by a variety of predicate structures each with a different combination of preconditions and effects. This means that there is more than one version of each pattern type, where each one has a different predicate structure with a different set of preconditions and effects, but the pattern type as determined by the set of concurrent sequences possible, is the same. POPI focusses on detecting each pattern type's predicate structure during the planner's preprocessing phase. The versions of the pattern

types that POPI detects are the ones that are catalogued and described in Chapter 3. At face value, it may seem that POPI only being able to detect one version of each pair-wise configuration is limited. However, these structures illustrated in Chapter 3 that POPI looks for in the domain are subset minimal, meaning that for each pattern, every predicate displayed is needed to maintain that unique pattern type's structure.

An interesting consequence of this is that, given the version of each pattern type that is detected and handled by POPI, it is possible for a pair of actions to be detected as being of more than one pattern type. This is because the predicate structure of some pattern types are subsumed by the predicate structures of other pattern types. Pattern E and J are an example of this, since pattern J contains all of the pattern facts of pattern E, facts $p$ and $q$, but also has predicate $r$ as an additional pattern fact that makes it more restrictive than pattern E. When there is an action pair detected as being of more than one pattern type, the planner needs to use the inference associated with the pattern type, that has the most predicates in its structure, in order to execute the correct inference for that pair of actions. This may seem counter-intuitive, as it seems that it should always be the detected pattern type with strongest inference that is fired. However, this could only be done if there is a guarantee that all pattern types subsuming the structure of another always have an associated set of pattern sequences with a stricter set of ordering constraints with less flexibility. However, this is not guaranteed and it must be that the pattern type detected with the most pattern predicates, is the one for which the inference is fired. The pattern structure with more predicates takes precedence. This is used as a method for resolving pattern conflicts.

## 6.3 Domain Analysis

In this section we describe our approach for analysing domains to look for patterns of required concurrency. In temporal domains there are various segments of information that need to be extracted and appropriately stored such that the subsequent pattern detection process can occur. First, the domain analysis only occurs if it is a temporal domain containing durative actions. The implementation of POPI in its current version has been designed to terminate if the domain being analysed is a temporal domain with durative actions but also contains instantaneous actions as well. The reason for this is that we want to focus on performing the analysis on problems where all actions are embedded in time and all operators can be analysed for required concurrency.

## 6.4 Pattern Detection and Storage

### 6.4.1 Pattern Matching

Following the extraction of the relevant information from the domain structure needed to identify the patterns of required concurrency presented in section 3.2, pattern detection can occur. The purpose of the pattern detection is to create the pattern instances that store the pairs of operators with required concurrency between them, the parameter indexes that need to match for the grounded actions, the pattern type, and a unique pattern ID. Each occurrence of required concurrency being detected between a pair of operators has this set of information stored in a *pattern environment* data structure. Each of these pattern environments are then recorded in a globally accessible map the key for which is the trigger operator. This allows for fast indexing into the map of pattern environments, when a grounded trigger action that is an instance of the trigger operator is applicable in a state. POPI can go through applicable actions and determine which, if any, of the actions can be used to trigger a pattern enabling the use of inference.

There are 15 pattern types for which POPI does pattern detection. We describe how this occurs for the first pattern type; the detection for the other pattern types are variations of this algorithm, which differ depending on the number of predicates they have and the number of comparisons that need to be performed. Algorithm 4 shows the procedure used to detect when a pair of actions satisfies the prerequisites of required concurrency pattern A, shown in Figure 3.3. The other patterns in Figures 3.4 to 3.18 follow the same type of logic for the detection of their respective concurrency patterns. The difference between them is the comparison made between different sets of candidate predicates and the actions linked to them. We note that in principle it is possible to develop an alternative algorithm which is able to compare all of the relevant lists of candidate predicates and operators for all pattern types, such that the implementation of this algorithm can detect any of the pattern types that we handle. However, in the current version of POPI described in this thesis, we implement variations of Algorithm 4 for the detection of each pattern type individually.

### 6.4.2 Parameter Index Matching

Required concurrency is matched between pairs of operators during a domain analysis that takes place before operators are instantiated using objects from the problem instance into grounded actions. Patterns are matched using a set of predicate(s) for each pattern type. Some parameters of one or both actions in the pattern may be for non-pattern fact predicate(s) which are also part of the preconditions and effects of the actions. This means that there could potentially be many grounded actions that could be used as the action to satisfy the constraints of the required concurrency relationship, if the grounded trigger action is added to

**Algorithm 4:** matchPatternA

```
Data: domainAnalysisMapStructures
Result: PatternEnvironment
1  foreach proposition p1 in candidateStartAddEffects do
2      if p1.numOfOccurences != 3 then
3          continue;
4      foreach proposition p2 in candidateEndDeleteEffects do
5          if p1.predicate == p2.predicate && p1.args == p2.args then
6              foreach proposition p3 in candidateOverallConditions do
7                  if (p1.predicate == p3.predicate) then
8                      foreach action a1 in candidateStartAddEffects[p1] do
9                          foreach action a2 in candidateEndDeleteEffects[p2] do
10                             if a1 == a2 then
11                                 temporalEnvelopeExists;
12                                 foreach action a3 in candidateOverallConditions[p3] do
13                                     if a2 != a3 then
14                                         return patternEnvironment (a2, a3, typeA);
```

the plan. If out of the available grounded actions that are instances of the inferred operator, only one of them is applicable, then this is chosen as the inferred action to apply. If more than one action is applicable, this means there is a choice about what action to use and inference cannot go ahead, therefore application of the pattern is not pursued.

### 6.4.3 Predicate Counting

Since the inference of POPI is based on the patterns detected during the pre-processing stage, it is important that only action pairs that are exact matches to the predicate structure of the pattern type being detected are recorded as instances of those pattern types. In addition, to make sure that those actions in the domain are actually required to be concurrent, we impose the strict rule that all the predicates that make up the pattern structure can only be achieved by the two actions that are candidates for the pattern. This is to ensure that there is required concurrency between the two actions. Furthermore, we also check that none of the predicates that are part of the two action schemas in the pattern have an instantiated literal true in the initial state. Although there could be required concurrency without these two restrictions, we enforce these as checks to provide certainty that there cannot be a sequential solution, if the planner chooses to use an action in a pattern. Therefore if the number of occurrences each predicate that is a part of the pattern structure and the initial state of the problem instance, is not equal to the number of occurrences in the pattern type as shown in the pattern diagrams in Figures 3.3 to 3.18 in section 3.2.2 of Chapter 3, then it is discarded as a candidate pattern instance. The check that the number of predicate occurrences is as specified for each pattern type occurs in its pattern matching method. An example of this

is shown in Algorithm 4, on lines one and two, which show that there can only be three occurrences of the predicate making up the required concurrency structure for pattern A. If this condition is not satisfied then the next candidate predicate is considered.

### 6.4.4 Recording IDs In Pattern Environments

We now discuss how POPI records only the list of action IDs that are candidates for being the inferred action of each pattern environment. In order to perform the inference faster when a pattern is triggered during search, the IDs for all the grounded actions, that are instantiations of the inferred operator, are stored in the pattern environment created for each successful pattern detection. Storing these IDs occurs just after all the actions have been grounded, which is the standard process of our baseline planner POPF. Algorithm 5 shows our method for collecting and storing the action IDs for the inferred operator component of each pattern environment created during the pattern matching phase.

**Algorithm 5:** storeInferableActIDs

```
   Data: groundedActions, globalPatternsMap
   Result: storedIDs
1  foreach action in groundedActions do
2      groundedAct ← getGroundedActionID[action]
3      foreach patternMapsList in globalPatternsMap do
4          trigOp ← patternMapsList.key
5          foreach pattern in globalPatternsMap[triggerOp] do
6              inferredOp ← pattern.getInferredOp(trigOp)
7              if action.getOp() == inferredOp then
8                  pattern.inferredActIDs.insert(action.ID)
```

## 6.5 Multiple Detections

It is possible for pairs of actions to be detected as being of more than one pattern type from the cases that we handle. Currently, the way this is handled for this situation is that a pattern environment instance is created for the detection of each pattern type. However, POPI has been designed so that for the pattern types, whose predicate structure subsumes another, it is that pattern type whose detection is matched first. The first mapped pattern instance is the one that is triggered during planning, so the correct pattern type is triggered in this situation. For example, when a pattern H is detected, the action pair will also be detected as being of Pattern M. The predicate structure of pattern H subsumes that of type M, and type H cases are matched first when the planner looks for pattern matches after the domain analysis. Therefore, any triggering of an action pair $a$ and $b$ in this scenario is correctly triggered using pattern type H. Alternatively, if $a$ and $b$ are detected as being of multiple pattern types, where the pattern types are disconnected and the predicate structure of one

does not subsume that of the others, the pattern type of the first detection is the one used to trigger the inference. It is important to note that in this situation the detection of the required concurrency according to all the pattern types detected are correct, and that the other and or additional predicate(s) making up the structure of another pattern type are effectively external predicates from the structure of the pattern type being used for triggering inference. If one of those predicates or any other predicate external to the pattern type causes a pattern action to not be applicable at a state, then the application of the pattern of actions as a whole is abandoned. Neither of these situations are an issue where a pair of operators is only detected as being of one pattern type.

## 6.6 Inference Engine

This section describes and explains the process that POPI goes through to perform inference and to infer a new action when a pattern environment is triggered. We will explain how POPI adds the ordering constraints between the snap action components. The process of doing inference is the same for both the aggressive and passive strategies which are referred to as EHC-AI and EHC-PI respectively. There are two versions of POPI, the difference between them is that POPI-AI uses the EHC-AI strategy for pursuing the use of inference aggressively, while POPI-PI is the passive version that uses EHC-PI. The inference strategies are described later in the chapter. The inference engine is implemented in the "CheckForInference" method. It is a recursive algorithm and allows chains of pattern actions to be inferred by checking if an action inferred by triggering one pattern environment, also triggers another different pattern environment, allowing another action to be inferred in a chain. The length of the chain depends on how many actions in different pattern environments are linked together.

### 6.6.1 Used Patterns

In order to prevent one pattern environment from being fired again, while one instance of it is currently being executed, POPI records the pattern environments which are currently being used for inference. This is because POPI does not allow multiple instances of the same pattern environment to execute at the same time. Each time a pattern environment is triggered for inference, it is temporarily recorded as a "used pattern" environment. The inference is performed for that pattern environment and when all of the inferred actions for it have been applied, the combined infer and search algorithm of POPI, using either the aggressive or passive approach, that calls the inference engine, deletes the record for that pattern environment in the used patterns list. At this point the pattern environment can be triggered and used for inference again.

### 6.6.2 Inferring New Actions

If a pattern environment is triggered, there are some steps that must occur to ensure that the action being added to the plan is the only viable one for use, and that no other action can be applied to satisfy the preconditions of the trigger action. If not, this would provide a choice point about which specific action to use concurrently with the trigger action and we could no longer do inference. POPI determines whether there is one possible action that can be inferred as required in the plan, given the use of the trigger action. It does this by going through the action IDs that were stored in the pattern environment for the inferred operator, and checking if the arguments for the required parameter indexes match for grounded action pair. If there is only one viable grounded action that has the required parameter index matchings for the grounded trigger action, then this is the inferred action.

### 6.6.3 Adding Constraints to Inferred List

POPI records the constraints for each pattern type, which is then used to order the snap actions in the pattern. The inference engine is encoded with all the ordering constraints for each of the 15 pattern types that it handles. When a pattern environment is triggered, its pattern type is first determined, as is it the specific grounded action that is to be added to the plan via inference that needs to be ordered relative to the trigger action. The trigger action, which is one component of the pattern, and the inferred action are both added to a stack structure. This stack is then reordered according to the constraints of the pattern type for the pattern environment that has been triggered.

## 6.7 Inference Strategies

This section presents two approaches designed to integrate temporal inference with the search performed in Enforced-Hill Climbing (EHC). Two variants of a combined infer and search process are presented, one for an "aggressive" based inference approach and another for a "passive" based approach. Both strategies trigger patterns in the same way; the difference between the two strategies is how each determines when inference should be pursued over search. Both algorithms are modified versions of Enforced-Hill Climbing as it exists in POPF. The normal search strategy of EHC is used, until a trigger action for a pattern instance becomes applicable, at which point depending on the strategy, inference is employed. In addition the main module developed to perform the inference itself, used by both strategies, is explained.

Both strategies enforce action ordering amongst the set of applicable actions. Applicable actions that are triggers for pattern inference are prioritised, these are chosen from the helpful actions first. However if there is an applicable action that triggers inference that is non-helpful, because there is no action that is helpful that triggers a pattern, then the

non-helpful pattern trigger will be chosen. An important point about the inference based approaches, is that in certain cases it is able to bypass local search through a plateau. If a plateau is reached by the search machinery, it may be possible there exists a sequence of inferred actions that can navigate the planner out of that part of the state space.

As can be seen in Table 3.1 in Section 3.3 of Chapter 3, not all pattern cases result in the planner inferring the addition of a new action to the plan when triggered. Pattern cases which only allow the planner to become informed that required concurrency exists or only being able to infer additional temporal constraints, are limited in the inference they provide compared with what POPF already does. This is since POPF infers that any action that is started must also be ended. For the pattern trigger cases where both start actions are needed to enable the inference, POPF knows that both of the corresponding ends must also be applied. However, POPF does not know explicitly that this is due to there being required concurrency between actions. POPI knows there is required concurrency and in most of these cases also infers additional temporal constraints. Although since POPI is implemented as an extension of POPF, to check that both start actions are in the plan and then add the inferred temporal constraints is a cost without a significant gain. This is especially since POPF will implicitly learn of these inferred temporal constraints when checking when and in what order it can apply the ends of the actions. Most pattern types and their associated trigger cases have one trigger action and can infer the addition of a new previously unknown action. For this reason, the focus of the implementation of POPI is on the pattern types and trigger cases, where the corresponding temporal inference results in newly inferred actions. The inferences of patterns A, B and I are only utilised in a chain of patterns where action B is added as part of another pattern, and action $A$ can be inferred in a backward chaining inference. Pattern trigger cases $\mathrm{L}_{BA}$ and $\mathrm{O}_{BA}$ do not infer any new actions to the plan and therefore are not accommodated by this implementation of POPI.

We will now go through the main techniques employed in the infer and search approach that is common to both the aggressive and passive variants. First, only one instantiation of each pattern instance is allowed to exist and be applied via inference at a time.

There are two key benefits that combined infer and search strategies bring:

1. Less search required when the planner is at a state where a pattern trigger action is applicable, allowing temporal inference to be used and therefore avoiding search over other action choices and pruning the search space.

2. The intermediary states constructed along the path generated by a pattern of actions can be explored with certainty based on search only having chosen to apply the trigger action.

### 6.7.1 Aggressive Inference Approach

This section describes a modified version of the Enforced-Hill Climbing algorithm, originally presented by Hoffmann and Nebel [2001], for the aggressive approach to temporal inference, which we refer to as Enforced-Hill Climbing with Aggressive Inference (EHC-AI). The logic for this modified version of EHC is presented in Algorithm 6. In this approach when the planner is in a state $S$, if an applicable action $A_{\vdash}$ is a part of a pattern instance and triggers inference, then all of the actions in pattern $P$, which $A_{\vdash}$ is a part of, are successively applied to reach the state at the end of the inference path, $S'$. At this moment, none of these pattern actions and the states produced from applying them are committed to. The planner has speculatively applied the actions in the pattern instance, that has resulted in a series of state generations. If the heuristic of $S'$ at the end of the inference, is lower than that of $S$, where the pattern instance was triggered, then the planner commits to the pattern actions and progresses to $S'$ as the new current state. If the heuristic of $S'$ is the same or worse than that of $S$, then the planner does not commit to the pattern actions and instead generates alternative successor states using an alternative pattern of actions, if it exists, otherwise a helpful action is selected using the standard approach of EHC as is done in POPF.

A key benefit of this algorithm is that commitment to the pattern of actions is held off, until the state $S'$ is reached and is evaluated to be better than the heuristic of state $S$ where the pattern instance was triggered. This is of significant benefit where $S'$ is a dead-end state. If this happens, the planner has recorded the state $S$ as the current state, and therefore will continue state progression from $S$ and not $S'$. The planner will navigate down a different path in the search space from $S$, still in EHC-AI mode. This allows POPI to recover from inference generated dead-end states, when using the EHC-AI strategy.

The idea behind the approach for pursuing inference aggressively, is that doing the inference allows the planner to avoid performing search over a set of action choices. The planner generates the states from the actions in the pattern or chain of pattern instances and only compares the heuristics of the state before the inference and after it.

#### Aggressive Inference Resulting in Better End Successor

In Figure 6.1 we see that at state s0, a trigger action in a pattern instance, $A_{\vdash}$, has become applicable. The planner does inference and finds that is must also add actions $B_{\vdash}$, $B_{\dashv}$ and $A_{\dashv}$, if it is to add $A_{\vdash}$ to the plan. All four pattern actions are successively applied to generate states s1, s2, s3 and s4, which is the resulting state at the end of the inference. State s4 is evaluated and has a better heuristic than s0, which is where the pattern instance was triggered. The planner commits to the pattern actions and progresses to s4, and carries on with search from here.

**Algorithm 6:** EHC-Aggressive-Inference (EHC-AI)

```
   Inout    : States I, S, S', helpfulActions, applicableActions
 1 evaluate I;
 2 S ← I;
 3 if S is dead-end then
 4     return problem unsolvable;
 5 if S is goal then
 6     return plan solution;
 7 else
 8     open_list.push_back(S);
 9 while !(open_list.empty()) do
10     S ← open_list.pop_front();
11     reorderHelpfulFirst(applicableActions, helpfulActions);
12     foreach a in applicableActions do
13         if a.patternTrigger() then
14             checkForInference(a, S);
15             if !(inferredActionsList.isEmpty()) then
16                 foreach pa in inferredActionsList do
17                     apply(pa, S) → S';
18             evaluate S';
19             if S'.heuristic == 0 then
20                 return planSolution
21             else if S'.heuristic < S.heuristic then
22                 open_list.clear();
23                 open_list.push_back(S');
24                 break;
25             else if S'.heuristic >= S.heuristic then
26                 discard S';
27                 break;
28     else
29         apply standard EHC in Algorithm 1
```

### Aggressive Inference Resulting in No Better End Successor

Figure 6.2 illustrates a situation where the pattern actions are applied, generating states s1 to s4, however when the value of s4 is assessed by state evaluation, it is worse than the heuristic of s0. The planner has so far only committed to state s0 in the state progression, therefore it does not commit to the pattern actions and does not progress down this part of the search space. Instead it uses search from s0, generating another child state out of s0, using a helpful action, $K_{\vdash}$ and progressing to state s5.

### Aggressive Inference Resulting in Dead-end

Figure 6.3 illustrates a situation where the pattern actions are applied, generating states s1 to s4, however this time s4 is evaluated as being a dead-end. As none of the pattern actions have been committed to, the planner remains at s0 and assigns a value of -1 as the heuristic

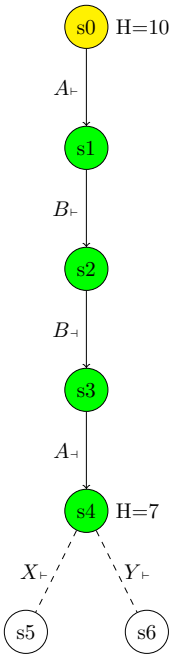



Figure 6.1: Aggressive Inference leads to a lower heuristic state.

of s1, since it is now known that the inference from this state results in a dead-end. As in the case of a worse heuristic situation shown in Figure 6.2, the planner again chooses a standard helpful action to continue down an alternative path in the search space. We see that action K⊢ is chosen to progress to state s5 which is closer to the goal.

### 6.7.2 Passive Inference Approach

In this section the passive based inference approach is presented. The algorithm for the modified version of EHC, implementing the passive inference approach (EHC-PI) is presented in Algorithm 7. A key difference is that instead applying all of the pattern actions in the *inferredActionsList* without evaluating any of the intermediary states produced, the passive approach evaluates the first state generated by the trigger action, which is part of the pattern. Only if this is better than the heuristic of the state where the trigger action was selected, does POPI apply the rest of the pattern actions without evaluating any of the remaining intermediary states. The other key difference is that if the heuristic of the state produced by the last action in the pattern is worse that the state from where the trigger action was applied, instead of going back to that starting state like aggressive strategy, EHC-PI fails and POPI goes into Best-first search. The difference between the aggressive and passive approach is in how the planner chooses whether or not to pursue the inference and when the actions in the pattern instance are committed to being in the plan. The time of commitment to an action is a key decision point, since a fundamental principle of EHC is that it always moves

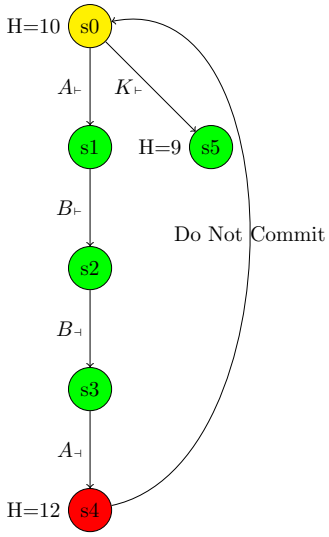



Figure 6.2: Aggressive pursuit of Inference leads to a state that is no better.

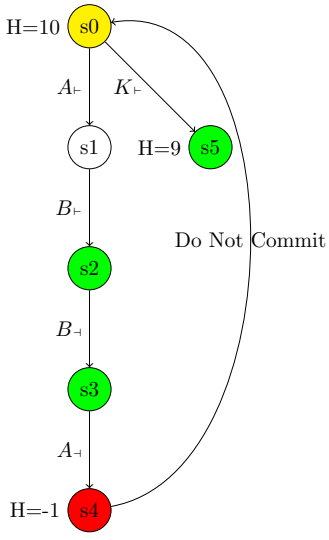



Figure 6.3: Aggressive pursuit of Inference leads to a state that is a dead-end. State s1 is assigned a heuristic of -1.

forward, and does not backtrack out of decisions it has committed to, which is a property maintained by both EHC-AI and EHC-PI. Suppose the planner is at a current state $S$, where a trigger action, $A_{\vdash}$, is applicable. This action is used to expand the child state, $S'$, which is not yet committed to. If $S'$ has a lower heuristic than $S$ as evaluated by the pre-existing TRPG state evaluation mechanism, then $S'$ is progressed to, and action $A_{\vdash}$ is committed to. Following the commitment to the trigger action, which is the first part of the pattern, the

rest of the pattern actions are applied. If all of the actions in the pattern are applicable as state progression occurs and are applied successfully, then the resulting state $S''$ produced at the end of the inference is evaluated. If the heuristic value of $S''$ is lower than that of state $S'$, then the planner progresses to $S''$. If state $S''$ is evaluated as having either the same or a higher heuristic than $S'$, then EHC-PI fails. POPI then resorts to best-first search, starting at the initial state as POPF does if EHC fails. The same will happen if POPI reaches $S''$ and it is evaluated as being a dead-end after committing to using the trigger action. The reason is that in both of these cases, POPI has already committed to part of the pattern being applied, and does not backtrack out of that decision. In this situation because POPI has committed to a state with a better heuristic, if it then applies the rest of the pattern and it finds a dead-end after applying any of the remaining pattern actions, it knows that there is no way to recover and no point attempting to apply other applicable actions between the state where the trigger action was applied and before the state where the pattern action resulting in the dead-end was reached. Search across a plateau as POPF would do in EHC would not help and this is the reason EHC-PI fails over to best-first search, to avoid unnecessary search.

#### Passive Inference Resulting in Better End Successor

The example shown in Figure 6.4 displays a situation where the action $A_{\vdash}$ is applicable at state s0, which is then applied resulting in state s1. The state s1 is evaluated with having a heuristic of 9, which is better than s0's heuristic of 10. The planner commits to $A_{\vdash}$ and progresses to s1. The actions $B_{\vdash}$, $B_{\dashv}$ and $A_{\dashv}$ are inferred and applied resulting in states s2, s3 and s4. The heuristic of s4 is better than s1, therefore the inference has been beneficial and search may continue. The difference with this example compared with the example using EHC-AI shown in Figure 6.1, is that the heuristic of s4 in the passive approach must be better than the heuristic of s1 and not s0, as is the case in the aggressive approach. This is because in the EHC-AI example in Figure 6.1, s1 is not committed to until the full overall value of the pattern application has been determined by evaluating the resultant state at the end of the inference, s4.

#### Passive Inference Leads to Same or Worse End Successor

Figure 6.5 presents the case where, the planner is at state s3 as the current state. Here, an applicable trigger action $A_{\vdash}$ is applied resulting in state s4, which is then evaluated. State s4 has a lower heuristic, so the planner now progresses to s4 and commits to it as the new current state. Following this, the rest of the pattern actions, for which $A_{\vdash}$ is a part of, are applied via inference in order to satisfy the required concurrency temporal constraints. The state s7 produced at the end of the inference has a higher heuristic than s4, which is still the current state. Therefore EHC-PI fails and POPI resorts to best-first search. The benefit of EHC-PI over standard EHC, is that the planner knows that since a set of actions in a pattern are

**Algorithm 7:** EHC-Passive-Inference (EHC-PI)

```
   Inout    : States I, S, S', helpfulActions, applicableActions
 1 evaluate I;
 2 S ← I;
 3 if S is dead-end then
 4     return problem unsolvable;
 5 if S is goal then
 6     return plan solution;
 7 else
 8     open_list.push_back(S);
 9 while !(open_list.empty()) do
10     S ← open_list.pop_front();
11     reorderHelpfulFirst(applicableActions, helpfulActions);
12     foreach a in applicableActions do
13         if a.patternTrigger() then
14             checkForInference(a, S);
15             if !(inferredActionsList.isEmpty()) then
16                 apply (inferredActionsList.front() S) → S';
17                 inferredActionsList.pop_front();
18                 if S'.heuristic < S then
19                     foreach pa in inferredActionsList do
20                         apply(pa, S) → S';
21                 else
22                     break;
23             evaluate S';
24             if S'.heuristic == 0 then
25                 return planSolution
26             else if S'.heuristic < S.heuristic then
27                 open_list.clear();
28                 open_list.push_back(S');
29                 break;
30             else if S'.heuristic >= S.heuristic then
31                 discard S';
32                 terminate;
33     else
34         apply standard EHC in Algorithm 1
```

active, there are no alternative actions that can be applied instead of the remaining pattern actions, at states s6, s5 or s4. This is because once $A_{\vdash}$ has been committed and the planner has progressed to s4, any alternative local search from states s6, s5 or s4 is futile, as selecting alternative actions would not satisfy the required concurrency temporal constraints that exist within the pattern structure. The search for alternative actions from states s6, s5 and s4 that would be performed by POPF using standard EHC are avoided.

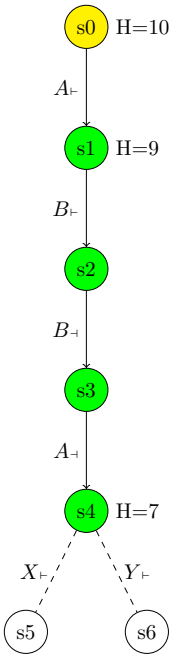



Figure 6.4: Passive Inference results in a lower heuristic state.

### Passive Inference Resulting in Dead-end

In the example shown in Figure 6.6, POPI is again currently at state s3, and commits to the pattern trigger action A⊢ and the resulting state s4, because s4 is evaluated as being better than s3. The required concurrency constraints necessitate that B⊢, B⊣ and A⊣ be in the plan resulting in the state s7, however state s7 is evaluated as being a dead-end. The forward moving principle of EHC, and their extensions EHC-AI and EHC-PI, mean that there is no back-tracking out of actions and states that have be committed to. For this reason, the commitment to A⊢ in s4 proves to have been a bad decision, hence at this point POPI terminates EHC-PI and resorts to performing best-first search from the initial state.

### Passive Inference where Pattern is Not Applied

The benefit of EHC-PI over EHC-AI, is that in cases where the first action (pattern trigger) applied from a pattern instance results in a state that is further from the goal, POPI will not commit to the application of the pattern and the sequence of states generated from it. Instead the planner will consider triggering alternative pattern instances if they exist, and if not then will use a non-pattern helpful action to progress through the search space, using standard EHC search mechanism. Using EHC-PI the planner only commits to the application of the pattern actions if the first successor has an immediate improvement in its distance to the goal.

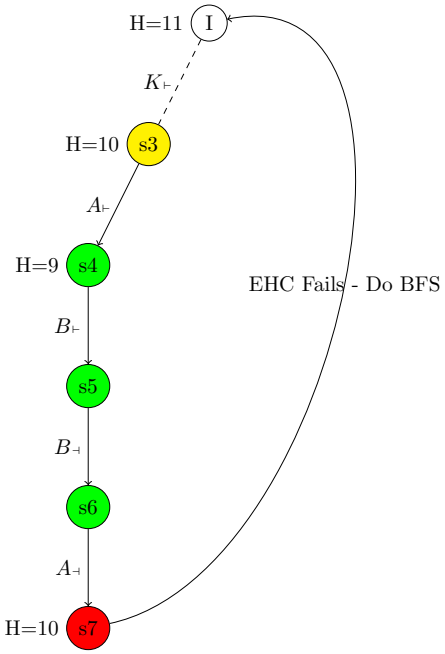



Figure 6.5: Passive approach to Inference results in state that is no better than or worse than current state.

We can see that the planner benefits from this approach in the example in Figure 6.7. Here, the planner is currently at s0 and the action $A_{\vdash}$ which is part of a pattern is applied and s1 is generated. s1 is estimated to be further from the goal state than s0. Therefore POPI does not commit to $A_{\vdash}$ and the other remaining actions in the pattern instance. In this example there is no alternative pattern that is applicable at s0, therefore the planner uses a standard non-pattern helpful action to progress search as standard EHC would do, the action $Y_{\vdash}$. It is true that if a better heuristic state exists at the end of the pattern application, then an opportunity to avoid some search and quicker state progression is then missed. However, this is the reason for the EHC-AI strategy, which has been developed to apply actions in a pattern more optimistically, and pursue the opportunity to use inference more aggressively. The passive approach is more cautious about applying patterns of actions.

### 6.7.3 Pattern Action Not Applicable

If a pattern action is not applicable, then for both the aggressive and passive strategies, POPI comes out of the application of the pattern and takes an alternative path from the state where the trigger was applied. The reason we do this for the passive approach as well as the aggressive, is because actions in a pattern are a single unit, if one is not applicable, then they are all treated as if they were not applicable. It is also to prevent an unnecessary

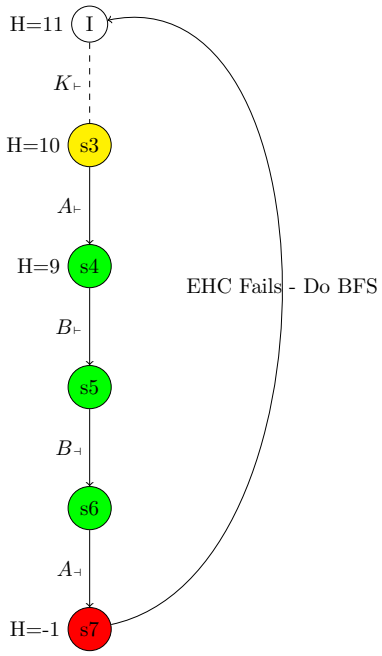



Figure 6.6: Passive pursuit of Inference leads to a state that is a dead-end.

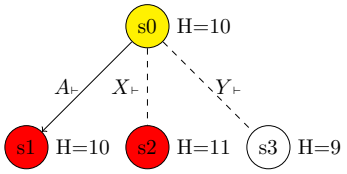



Figure 6.7: Passive approach - trigger action results in worse heuristic or dead-end, pattern not applied and commited to.

search, attempting to apply other non-pattern actions that might satisfy the non-pattern fact preconditions of the action that couldn't be applied.

## 6.8 Example Planner Output

In this section, example plans and the associated search space diagrams are presented for each of the aggressive and passive strategy situations that POPI can handle. We have designed a domain called `mybuilding` to illustrate the theory of the aggressive and passive inference strategies presented in sections 6.7.1 and 6.7.2 respectively. The `myBuilding` domain is in appendix D. It should be noted that in order show what the behaviour of each strategy is in the various scenarios, we modify the preconditions and effects of the actions in the pattern

structure, to trigger the mechanism in place for each of these scenarios to showcase the differences in behaviour. The `myBuilding` domain describes a building environment with a series of rooms. Rooms are connected with doors which must be open in order to go directly from one room to the other. Alternatively an action that makes a hole in the wall between two rooms can be performed allowing for travel between the two rooms, however this causes the building to no longer exist at the end of it. The initial state consists of being at the starting location and the goal is to be at the destination with the building still to be stable at the end.

In addition to displaying example POPI output for the different scenarios, according to the strategy being used, the examples in this section also show what happens when the predicate counting mechanism in the pattern detection is disabled. The method for counting predicates described in section 6.4.3 ensures that the number of times each predicate part of the pattern structure for each pattern type is exactly the number specified for that pattern type. The counting also checks the initial state and includes it in its total. If the count is not equal to the number specified for a pattern type, that pair of actions is discounted as being a pattern since required concurrency between them of that pattern type can not be guaranteed. This thesis focuses on inference, driven by certainty in knowing the pattern type of a pair of actions and that the required concurrency that it describes does indeed exist. However, the cost paid for this, is that where our inference approach could have been used, given the existence of a pattern, it is discarded due to the rules for counting predicates.

In the search space diagrams generated for the plans generated by POPI, the label 'O' in the nodes represent the order of node expansion and 'H' represents the heuristic of the state. In the cases where there is more than one number for 'O', each number represents the order number in which another branch out of that node is explored. If 'O' is -1, this means that no child nodes are expanded from it.

### 6.8.1 Aggressive Cases

#### Aggressive Inference Leads Closer to Goal

We see in Figure 6.8 the situation where after having applied two actions via the normal search machinery, POPI detects a pattern trigger, (make-hole room1 dest) (start), that triggers the use for inference. All of the pattern actions are applied successively, ignoring the heuristic values of each state generated during the application of the pattern actions up until the last one. This example shows that the heuristic of the last state in the search space is 0, meaning it is also the goal state. All actions that are started must also be finished, except in the case of compression-safe actions, where a separate state generation for action ends is not necessary. The search space diagrams show the application of an action's start, but not its end as it is

compression safe.

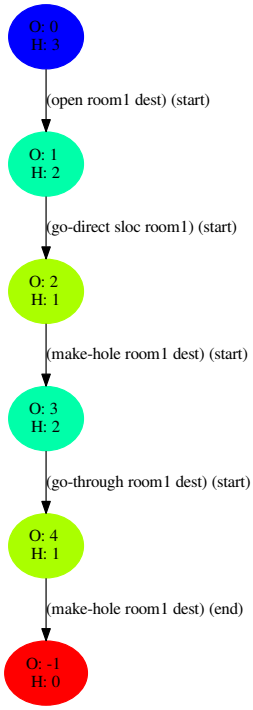



Figure 6.8: POPI using aggressive inference strategy, where the pattern of actions successfully lead to a lower heuristic state - the goal.

### Aggressive Inference Leads to No Better Heuristic state

Figure 6.9 shows the use of inference and applying the pattern of actions leads to a state that is not estimated to be closer to the goal than the state where the pattern trigger action became applicable. Therefore this path of pattern actions is not committed to and the planner expands a second branch out of the trigger state using a standard helpful action, which in this case is the action (go-direct room1 dest) (start) that results in the goal being achieved.

### Aggressive Inference Leads to Dead-end

Figure 6.10 shows the situation where the planner again takes an aggressive approach to inference, ignoring the heuristic of state labelled $o = 3$ and the planner again chooses not to commit to the pattern of actions, but this time because the state reached after inference is a dead-end. It should be noted that in the case of dead-end states, unlike worse heuristic states, if a dead-end is detected after applying any pattern action, the pattern is immediately abandoned as there is no chance of there being a better heuristic state after applying all of

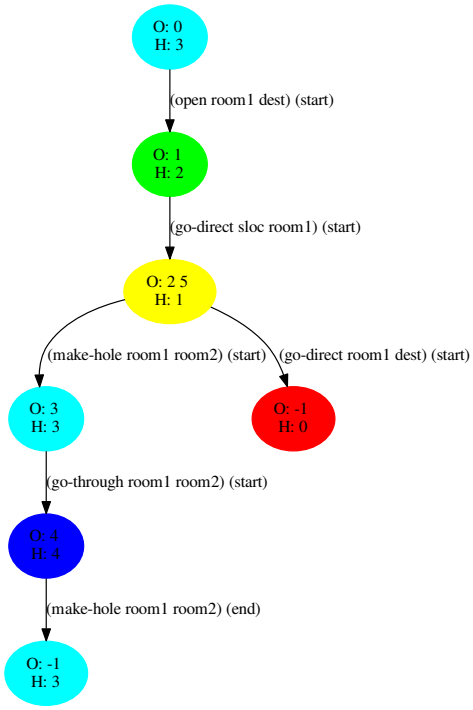



Figure 6.9: POPI using aggressive inference strategy, where inference leads to a no better heuristic state that is not committed to.

the pattern actions.

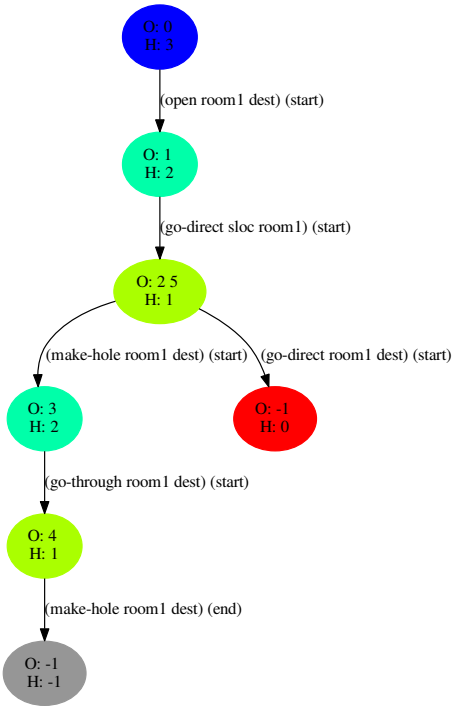



Figure 6.10: POPI using aggressive inference strategy, where inference leads to dead-end, that is recovered from.

### 6.8.2 Passive Cases

#### Passive Inference Leads Closer to Goal

In Figure 6.11 we see that the first state generated from the first pattern action produces a lower heuristic state, therefore this state is committed to for state progression, as it would be in standard EHC. Following this the inference takes place and the state produced after the pattern application has a lower heuristic and in this case has led to the goal itself.

#### Passive - Pattern Actions Not Applied

The case shown in Figure 6.12 shows a situation where according to the passive infer and search algorithm, EHC-PI, the first applicable action in the pattern instance is applied. The heuristics of the successor and the current states are compared. The remaining pattern actions are only applied if the heuristic of this first successor is lower than the current(trigger) state. Since the heuristic of the successor is higher, the planner does not commit to applying the trigger action and progressing to the successor. Therefore, none of the pattern actions are applied and the planner instead chooses an alternative helpful action, which in this case achieves the goal. The same happens if the first pattern action results in a dead-end state,

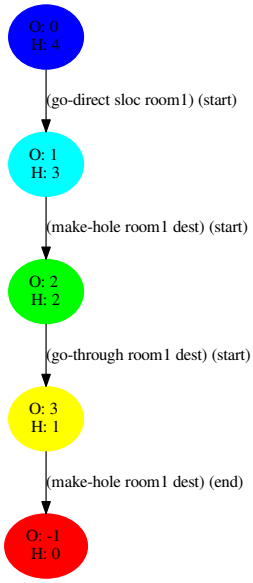



Figure 6.11: POPI using passive approach, where the pattern application and inference leads to a state closer to the goal.

since the the planner has not yet committed to using the pattern. An example of this is shown in Figure 6.13.

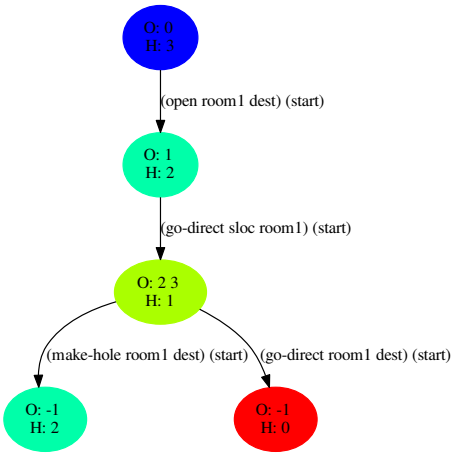



Figure 6.12: POPI using passive inference strategy, where pattern of actions are not applied because the trigger action results in a worse heuristic state.

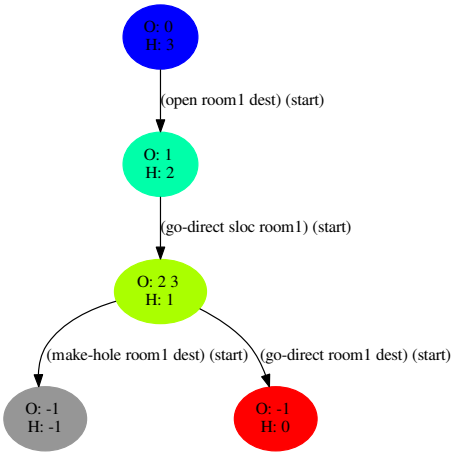



Figure 6.13: POPI using passive inference strategy, where pattern of actions are not applied because the the trigger action results in a dead-end state.

#### Passive Inference Leads to No Better Heuristic state

After applying all of the pattern actions, if the heuristic of the state reached is the same or worse than the state where the trigger action became applicable, then EHC-PI fails and the planner resorts to best-first search. The pattern of actions has been committed to and EHC-PI does not attempt to search through a plateau like EHC in POPF does, in order to find a better state, given that the planner may have to resort to best-first search anyway.

#### Passive Inference Leads to Dead-end

If the application of any of the remaining actions in the pattern after the trigger action result in a dead-end state, EHC-PI fails and POPI resorts to best-first search. This is because POPI knows that given it has committed to the trigger action of the pattern, all of the actions in the pattern must be applied or none of them at all due to the required concurrency relationship. EHC-PI does not to apply actions via search mid way through the application of pattern actions, in order to recover from the dead-end.

## 6.9 Completeness and Soundness

In this section we will discuss the aggressive and passive infer and search algorithms with respect to maintaining the properties of being sound and complete. The general approach for performing temporal inference is to leverage information that the planner acquires before search begins, during the state expansion phase. The planner POPI, which is extended from POPF, still resorts to best-first search if it fails to find a solution using its primary infer and

search strategy. As an extension of POPF, POPI can be considered complete if POPF is complete. This is because the plans that can be reached in EHC by POPF, and in EHC-AI and EHC-PI by POPI are a subset of the solutions that can be reached by using best-first search, which both planners resort to if the primary strategy fails to find a plan. *soundness* is where a planning strategy only produces plans which are valid. POPI maintains soundness in its solutions, since the triggering of a pattern and its resulting inference only adds actions which need to occur together, in the correct order for the detected pattern type. If an action in the pattern is not applicable, then the use of pattern is abandoned.

If the problem being solved contains required concurrency, where there is no sequential solution due to temporal constraints, the inference will only add actions that need to be concurrent given a detected pattern; this is with the predicate counting mechanism in place which is the default setting used. These actions are the same ones that the search mechanism would also need to add in order to achieve the goal. If the problem is one with optional concurrency, where there is a pattern of actions, POPI will attempt to apply them. However, if the inference leads to a worse heuristic state at the end of the inference or after applying the trigger action(s), for the aggressive and passive approaches respectively, the planner discards the use of these inference generated partial paths, and remains at the state before the trigger action was applied, so that an alternate route can be used. Again, POPI still maintains the use of best-first search, if EHC-AI or EHC-PI fail.

## 6.10 Summary

Although the method of checking and doing inference is the same, we can see from POPI's example output cases that the aggressive and passive variants of the infer and search process behave differently when it comes to choosing when to pursue the use of inference and when not to. The aggressive approach is more speculative and applies the actions in a pattern and assesses the value of the heuristic only at the end of the inference, whereas the passive approach will compare the heuristic value of the state produced from applying the first snap action in the pattern. Only if this first successor state produced from this is better than the current state, will it apply the rest of the pattern actions. These two algorithms utilise the core inference engine that tells the infer and search algorithm how to apply the pattern actions given the pattern type. Chapter 7 presents empirical results and evaluation of POPI using both the aggressive and passive inference strategies on domains containing both required concurrency and optional concurrency, where certain actions need to occur together for particular solutions to the problem, but where there are also sequential solutions.

# Chapter 7

# Empirical Analysis

In this chapter we discuss and analyse the empirical results produced by running POPI using both the aggressive and passive variants of its combined infer and search approach on domains with required concurrency and optional concurrency. In both cases, there will be pairs of actions that are in a pattern, but for optional concurrency problems there is more than one plan to reach the goal, at least one of which does not need to include the action pair in the pattern, since there is a sequential solution. The main purpose of the empirical analysis is to meet the following evaluation objectives:

1. To show that POPI-AI solves problems of required concurrency containing patterns faster than POPF and is able to scale better.

2. To show that other temporal planners are limited in their ability to solve required concurrency problems containing the patterns presented in this thesis.

3. To show that the overhead cost of both POPI variants on non-concurrent temporal problems does not prevent them from solving problems that POPF can solve.

4. To show that in general, the baseline planner POPF is able to compete with other temporal planners in solving temporal problems.

## 7.1 Planners for Experiments

The results for our experiments on POPI will be compared against the base planner POPF. The performance of POPI is also compared with POPF when run with a total ordering enforced, effectively giving us COLIN, the predecessor to POPF. Although inference is used in various forms in planning systems, we are interested in assessing the performance of POPI in its mechanisms to implement the use of more inference and less search. It is for this reason that we compare both variants of POPI against its predecessor, POPF. We will also run POPI with the partial ordering machinery disabled, where only total ordering is used during plan

construction. We refer to this version of the planner as COLIN-I, since POPF without partial ordering is COLIN. Partial ordering disabled in POPI means we effectively have our inference approach as an extension of COLIN rather than POPF. We will also analyse the benefits of POPI using the theory of pattern information content presented in Chapter 5.

In addition, as we are assessing the benefits of using inference, eCPT is also an interesting planner for comparison, since it uses constraint programming techniques for doing inference. However, we will see that not all temporal domains used for experiments in this chapter can be successfully handled by eCPT. Furthermore, we also use other temporal planners to compare against the performance of POPI. To that end, we perform experiments on temporal planners which took part in the temporal satisficing track of the most recent International Planning Competition (IPC) in 2018. There were five submissions in this track. From these planners two of them are systems which were extended from POPF, which are POPCORN and OPTIC. These planners introduce no additional capabilities for solving the types of required concurrency problems which this thesis investigates, hence we do not use them as planners for comparison in our experiments. For the purpose of solving our patterns, the behaviour of POPCORN and OPTIC is the same as the behaviour of POPF. TFLAP was another submission made to this track and is a temporal planner that does forward partial order planning; we test this system in our comparative analysis. The remaining two participating systems were CP4TP (Furelos-Blanco and Jonsson [2018]) and TemPorAl (Cenamor et al. [2018]). These are *portfolio* planners, meaning that they are each a combination of various planning systems, where each planner in the portfoilo attempts to solve a planning problem using only a segment of the allocated time. If one planner fails to solve the problem, it is passed on to the next planner in the portfolio. We are interested in comparing individual planning systems and therefore do not use these portfolios as comparators, but we do use some of their component planners which can handle required concurrency.

We will see though in our initial testing that these planners are limited in the types of required concurrency that they can handle. The component planners we test from CP4TP are TPSHE, TP($K$), for $K \in \{2, 3, 4\}$ and STP($K$), for $K \in \{2, 3, 4\}$. We also test the ITSAT and TFD planners from the TemPorAl portfolio. We present the results for TFD seperately to the other IPC planners for the tests on the patterns domains. This is because we will see that due to how the planner works in certain situations, it is not a planner that can be usefully and reliably be compared for all of our domains containing the various types of required concurrency. The remaining planners, YAHSP2 and YAHSP3 are also component planners from the TemPorAl portfolio, however due to compliation errors we were not able to run these planners.

## 7.2 Domains for Experiments

We focus the testing and experiments for our planning system on domains where actions are defined with duration inequalities as we are interested in the behaviour of our system compared with other planners where required concurrency of the pattern types presented in Chapter 3 exist. Some of these pattern types allow choice in the specific concurrent ordering of the action endpoints and fixed durations for some patterns would restrict the application of the pattern endpoints for some of those orderings.

Although there are existing domains that incorporate required concurrency amongst the benchmark domains, we have found that these domains contain concurrency of pattern types A and B which are not the most interesting in our analysis in terms of enabling powerful inferences. As illustrated in Chapter 3, patterns A, B and I, do not enable the addition of a new action to the plan via inference in the single pattern trigger cases. Historically, the common temporal domains that have required concurrency are those where there are temporal windows, as is the case in the `matchCellar` domain, which is an instance of pattern type A. Patterns A, B and I are among the weakest pattern types in terms of inferential power, because all three of these patterns have only one trigger case, where both the start of action $A$ and action $B$ must be in the plan to trigger inference. Without being part of a chain of patterns, none of these patterns enable the addition of a new action to the plan using our inference approach.

The purpose of POPI is to exploit the use of inference with speed to add actions and constraints, when it detects temporal structures that it can understand. Our approach is to also maintain the existing behaviour of POPF's previous problem solving strategies, when the required concurrency structures we handle, either do not exist in the domain or the pattern types detected cannot be exploited. Therefore, the fact that many available temporal domains do not contain required concurrency is beneficial to testing that POPI's behaviour is the same as POPF in these circumstances. To that end, we perform experiments using the temporal domains from the most recent International Planning Competition, IPC 2018, in order to test this aspect of POPI. The purpose of these experiments will be to confirm that the behaviour of POPF and both variations of POPI are the same in domains without our patterns of required concurrency. Experiments were ran on a machine using an Intel Core i7 Processor with a limit of 4GB of memory and 30 minutes of CPU time per problem. All graphs showing the results for time taken to solve problems are measured in seconds.

## 7.3 Required Concurrency Domains with Patterns

The set of `patterns` domains are an "artificial" set of domains that have been created specifically to test the capabilities of POPI and its performance when required to repeatedly apply a pattern of actions. These domains have also been constructed to allow us to see the behaviour of POPI in domains containing each pattern type that is handled, and to assess the scaling behaviour of POPI, comparing it against the theoretical information gain measurements seen in Chapter 5. Specifically, these domains showcase how POPI benefits from prioritising the use of pattern trigger actions to pursue inference over the standard helpful actions that are selected arbitrarily to generate successors. Since the `patterns` domains are used to demonstrate the inferential power of POPI, these domains are set up to present situations where the TRPG heuristic causes POPF, our baseline planner, to repeatedly apply an action that looks helpful, but is actually a bad decision. It presents a scenario where the heuristic guidance is misleading the planner about what action it should apply next and where POPI's approach can be more helpful.

The `patterns` domains have been defined using duration inequalities to provide bounded constraints on the minimum and maximum duration of each action rather than fixed durations. The reason for this is that it is possible in the case of certain pattern types such as pattern H, for one action to encapsulate the other and vice versa. For this reason we use duration inequalities to allow both orderings, and let the planners decide the durations and choose the order of applying the actions. This is because our comparison focusses on inference enabled by knowing the temporal structure of actions, not duration. By allowing the flexibility for planners to choose action durations, this provides the opportunity for all the possible orderings of the snap actions to be viable for all planners. This allows a comparison of the benefit from each pattern case's inference for all of the patterns with the other planners, which also have the same flexibility to choose action durations. As an example of the general structure of a `patterns` domain, the `patterns` domain containing the pattern type D structure is included in Appendix B. The other `patterns` domains are variations of this one. The PDDL for the action pair that makes up the pattern D structure in this domain is shown in Figure 7.1.

```
(:durative-action Act_A
 :parameters(?a ?b - typeA )
 :duration(and (<= ?duration 5) (>= ?duration 1))
 :condition(and (at start (active)) (at end (q ?a))
  (at start(next ?a ?b)) (at start (ready ?a)) (at end(ready ?a)) )
 :effect (and (at start(p ?a)) (at start(not(active))) (at start(ready ?b))
    (at end (active)) (at end(not(ready ?a)))
   )
 )

(:durative-action Act_B
 :parameters(?a  - typeA)
 :duration(and (<= ?duration 5) (>= ?duration 1))
 :condition(and (at start(p ?a)) (at start (ready ?a)))
 :effect (and (at end (q ?a)) )
)
```

Figure 7.1: Actions from `PatternsD` domain where `Act_A` and `Act_B` are in a pattern type D relationship.

### 7.3.1 POPI, POPF, COLIN and COLIN-I Experiments

In this section we perform an ablation study to test the performance of POPI against the baseline POPF and also against COLIN, the predecessor to POPF the core difference being that COLIN uses a total order during plan construction. We also the two variants of POPI using total ordering to gain an idea of how useful the partial ordering in POPI for problems in the `patterns` domains.

#### Results

This section describes and evaluates the results produced from running a set of experiments using the `patterns` domains. An individual version of the `patterns` domain modified to contain each pattern type structure has been run on both the aggressive and passive versions of POPI and POPF. In addition, all three of these planners have also been run using total ordering, effectively giving us COLIN with aggressive inference and COLIN with passive inference. However, we focus our analysis on POPI-AI (Aggressive strategy) versus POPF as our main comparison as we are interested is observing the scaling behaviour of POPI, which the aggressive strategy has been primarily been designed for, and we are interested in its performance over the base planner that POPI is built on, POPF. Although POPI practically evaluates all of the states it generates, it does not use the heuristic evaluations of the intermediary states generated via inference. Therefore, in the graphs presenting results for the states evaluated, the first 6 keys are for each of the planners as labelled but represent all the state evaluations performed. The last 4 keys are for the same POPI and COLIN-I planners, but the keys are appended with "SU" which stands for "States Used", and this is the figure which we are

interested in using for comparison. This is because it does not include the state evaluations which are ignored by the POPI or COLIN-I planners. Where the "States Used" value for our planners is on the upper most line, which represent the problem being unsolved by a planner, the planner can still be shown to have solved the problem according to its planner key for just state evaluations, without "SU". In this situation, it means that this planner solved the problem using best-first search, and did not solve it using its inference-based approach, either EHC-AI or EHC-PI.

For each of the Patterns domains, the first 10 problem instances go up in intervals of 10 objects in each subsequent problem instance, ranging from 10 to 100 objects. The subsequent 9 problems go up in intervals of 100 objects ranging from 200 to 1000 objects. We also run four larger problems with 1250, 1500, 1750 and 2000 objects, in order to determine the largest problem which any of the planners could solve across all of the Patterns domains. All of these domains contain required concurrency, where there is no sequential solution for any problem instance and only one plan to reach the goal. The only feature changed between each of the domains is the predicate structure of the action pair so that it conforms to the structure of each pattern type. We use the same set of problem instances for each of these domains. It is important to note that the patterns domains used in this set of experiments were specifically designed to showcase the situations in which POPI provides significant power in exploiting the structure of the pattern that exists in each domain.

For patterns A and B shown in Figures 7.2 and 7.3 respectively, we can see that POPI-AI and POPF perform the same, this is because patterns A and B do not enable the addition of a newly inferred action, meaning that although the patterns are detected by POPI, it defaults to the behaviour of standard POPF and solves these problems with the same number of state evaluations. This is also the case for the pattern I, shown in Figure 7.10. However, we can see that the number of problem instances solved for the pattern I domain is smaller. This could suggest that the ordering required for a pair of actions in a pattern I structure is more difficult to schedule and perhaps requires more memory, preventing larger problems from being solved. We can expect that there to be some gap between the theoretical information gain measurements for each pattern trigger case and the practical gain they each provide in practice compared to POPF. There is only one trigger case for patterns A, B and I, and a measurement of 1 bit of information gain of POPI over POPF assigned to each of them. This is because only one ordering constraint can be inferred over what POPF already knows about how the actions should appear in the plan when these actions occur together. In practical application, POPI does not explicitly add this inferred constraint, and so seeing that the number of problems solved by POPF and POPI-AI is the same, using the same number of state evaluations, is not a surprising result.

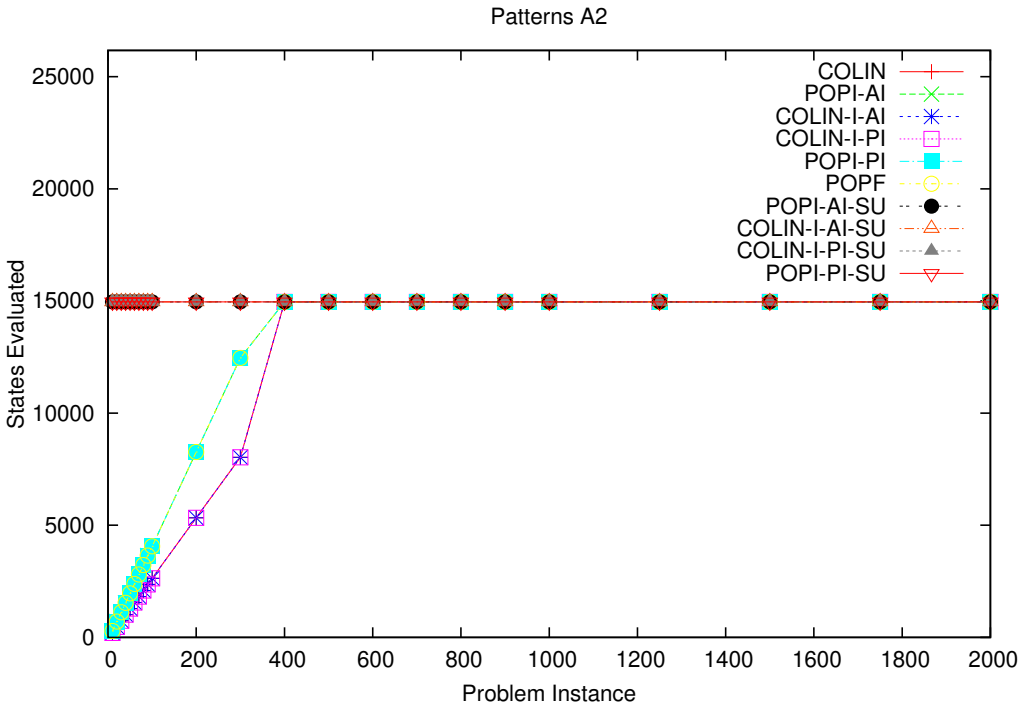
For the domain containing the pattern C structure shown in Figure 7.4, we see that POPI-AI scales well, solving problems with up to 1000 objects, requiring 499 repeated applications of the pattern. POPI performs the same in domains with pattern types D and E, solving the same number of problems instances, however the solution time of POPI-AI in the patterns C domain is much lower compared to its solution time for the pattern D and E domains. Once again POPF struggles to scale to the larger size problems as it attempts to repeatedly apply actions outside of the pattern, which the heuristic estimates is the best choice, but is in fact not going to take it closer to the goal. All of the other domains with single patterns, we see that a similar number of problems are solved by POPI-AI, where the solution time varies a relatively small amount. POPF consistently struggles to solve problems using these domains, the reason being is that its choices are being guided by its heuristic, which is consistently giving it the wrong action to apply.

Apart from Patterns A, B and I, each of the other 12 patterns has at least one trigger case where it enables the addition of a new action to the plan. For these cases, the calculated information gain of POPI-AI over POPF is 2 bits. All of the patterns domains for these pattern types were set up to trigger for these cases, such that we could compare the performance gain from the empirical results to the theoretical information gains. It is quite apparent that POPI's information gain over POPF, in the current scaling behaviour across the different domains, is not as high as we would have expected, given the theoretical calculation for each pattern application. With the scaling behaviour that has been observed, we could have expected a linear growth in the information gain, given the number of pattern applied for each problem instance. However, this is not the case because POPF does not explore the states generated from the pattern actions in a naive, compound manner, but it is not fully independent either. Although we would expect to see a gain of 2 bits in information per pattern in the scaling case, we see that the information gain grows slower than $n$ x 2 bits for n patterns.

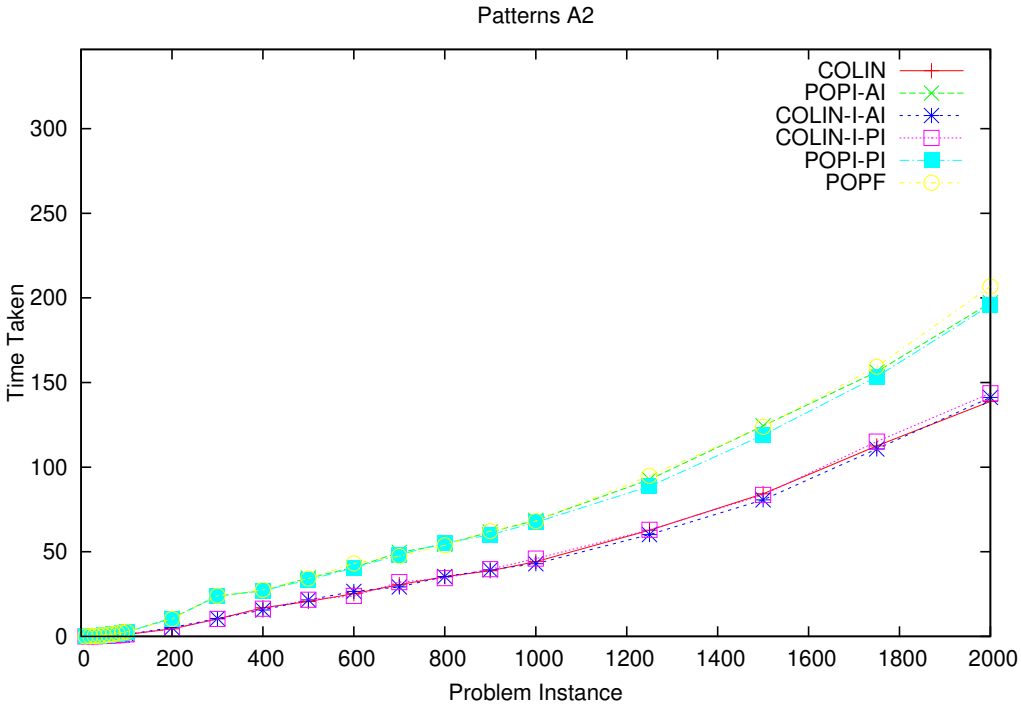
For the chained pattern domains presented in Figures 7.17-7.22, we can see that even POPI-AI has difficulty in solving some of the larger problems that it could solve in the single pattern domains. For these domains, both for the problems it could and could not solve, we see that the time taken curves have a shallow time growth, meaning that it consumed the memory very quickly. This applies to the time taken curves for all of the planners across all of these domains, where such curves exist. If the time taken curve is steep, this means that it takes the planner a long time to consume all of the memory available to it.

**Summary**

Despite the gains that POPI-AI has in solving larger scale problems, we can see that in all of these domains, none of the planners are able to solve the very largest problems containing 1500, 1750 and 2000 objects. This shows us that no matter which strategy is being used, all of these planners are limited, but to different extents. We have observed that POPI-AI does generally performs the best across all of the domains as would be expected for domains, where repeatedly applying these patterns of actions brings the planner closer. The results show us that generally POPI-AI is indeed able to solve problems of required concurrency containing patterns, faster than POPF and also scales better to solve larger problem instances. This allows us to satisfy the first evaluation objective listed at the beginning of this chapter.

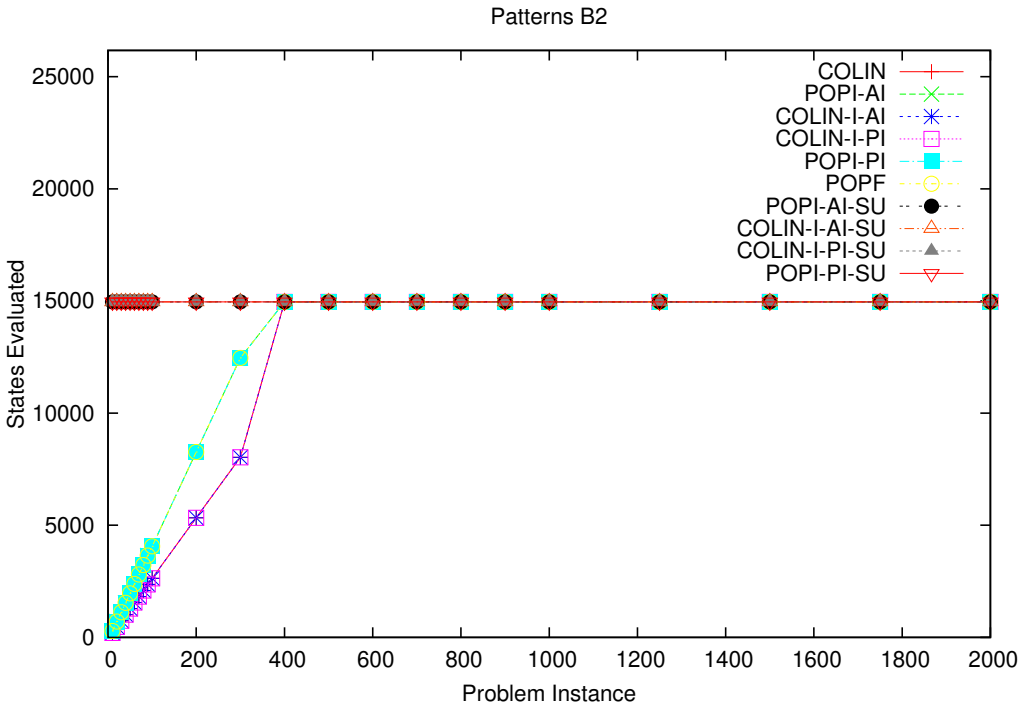



(a) Patterns A - States Evaluated

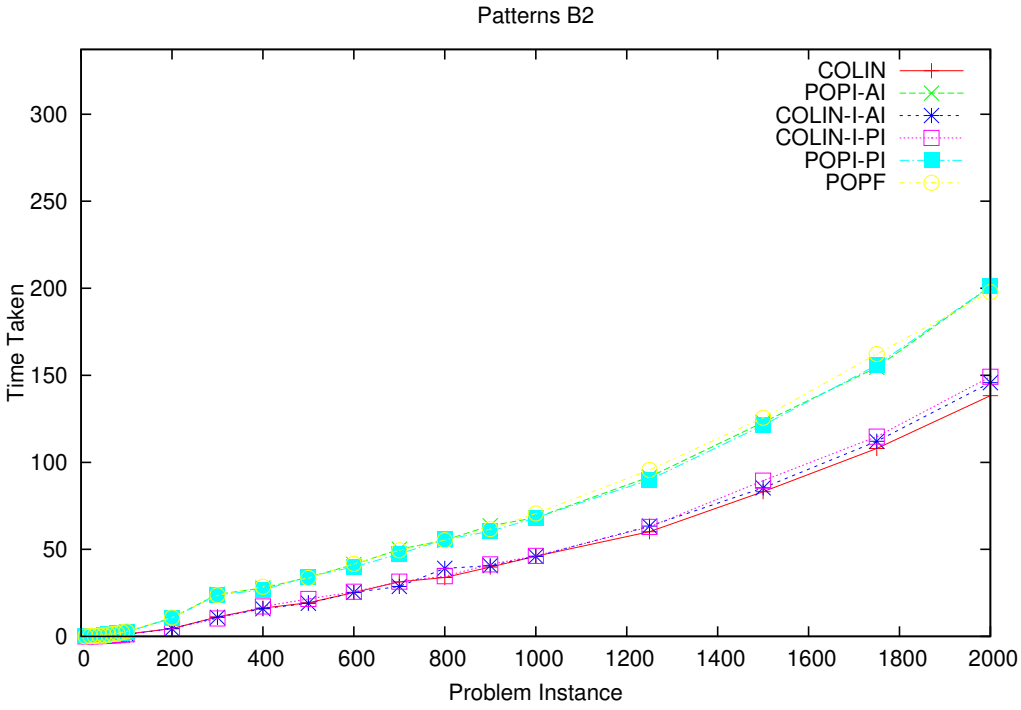



(b) Patterns A - Time Taken

Figure 7.2: Pattern A

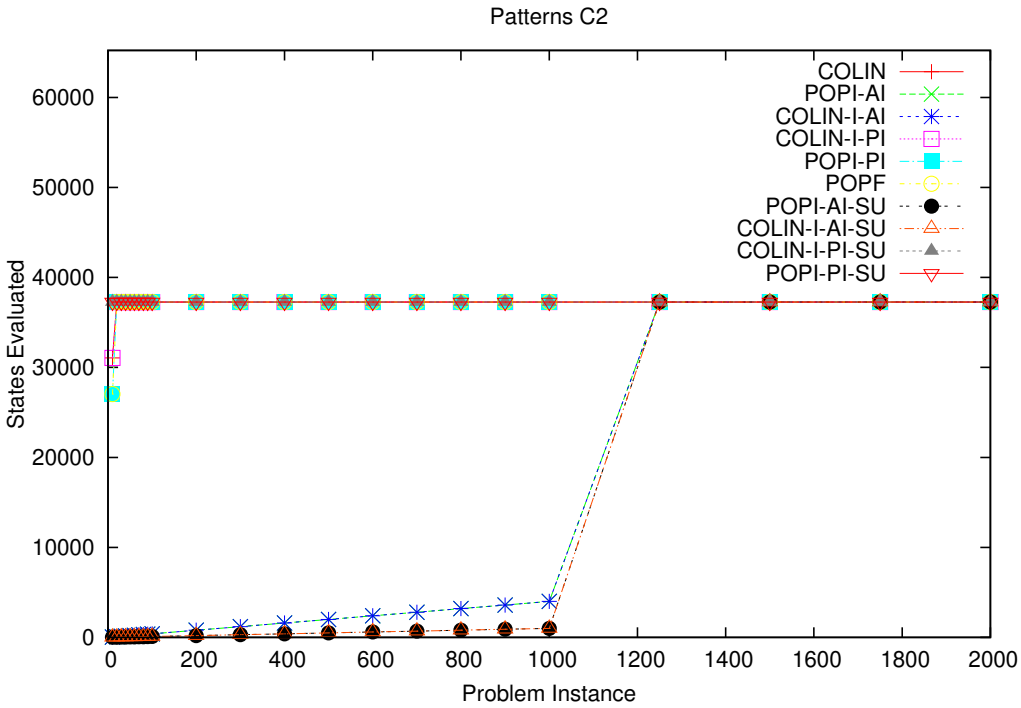



(a) Patterns B - States Evaluated

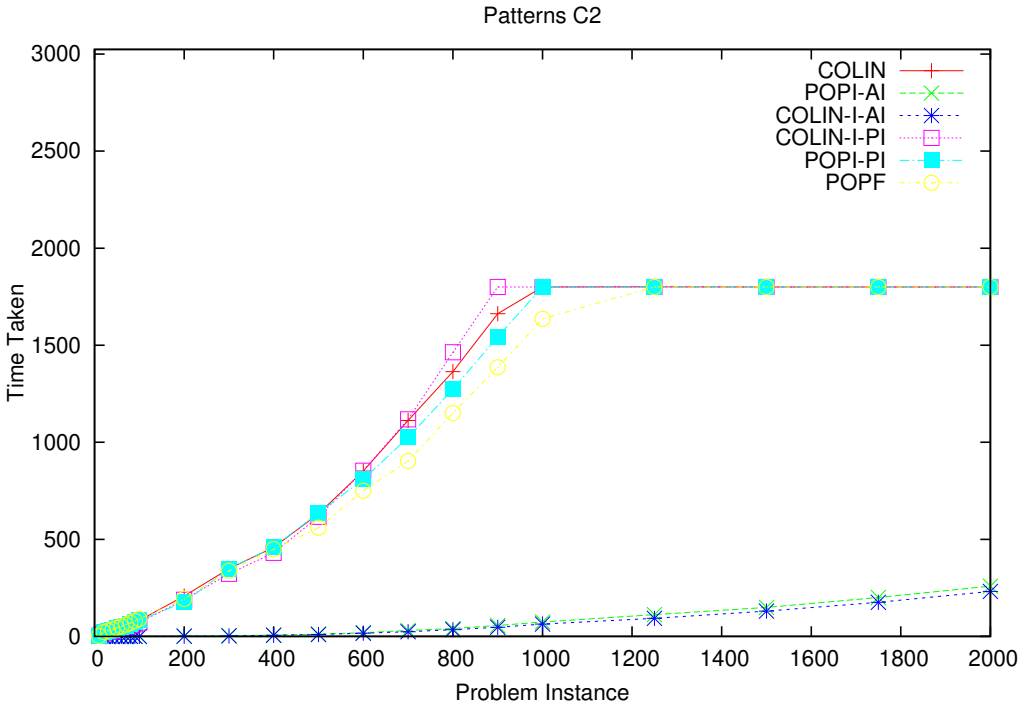



(b) Patterns B - Time Taken

Figure 7.3: Pattern B

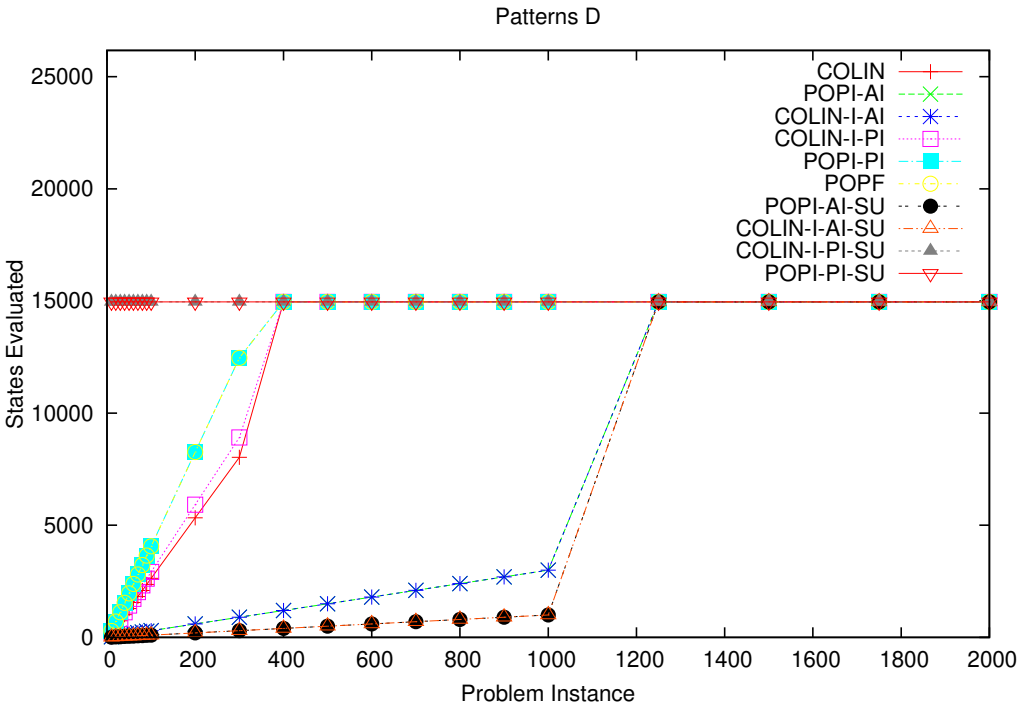



(a) Patterns C - States Evaluated

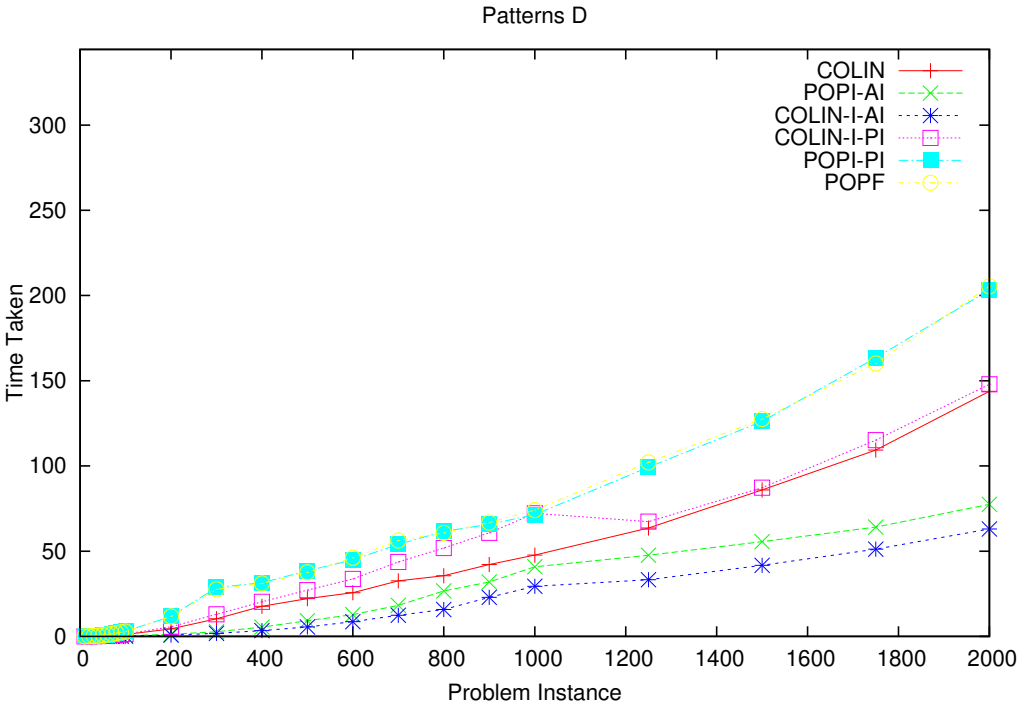



(b) Patterns C - Time Taken

Figure 7.4: Pattern C

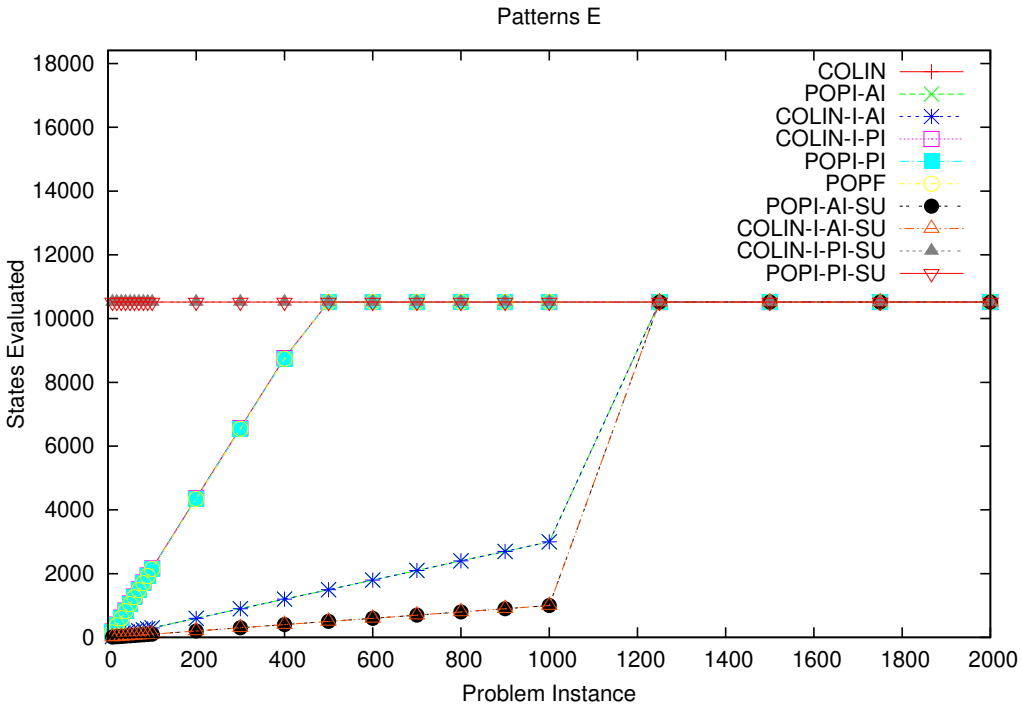



(a) Patterns D - States Evaluated

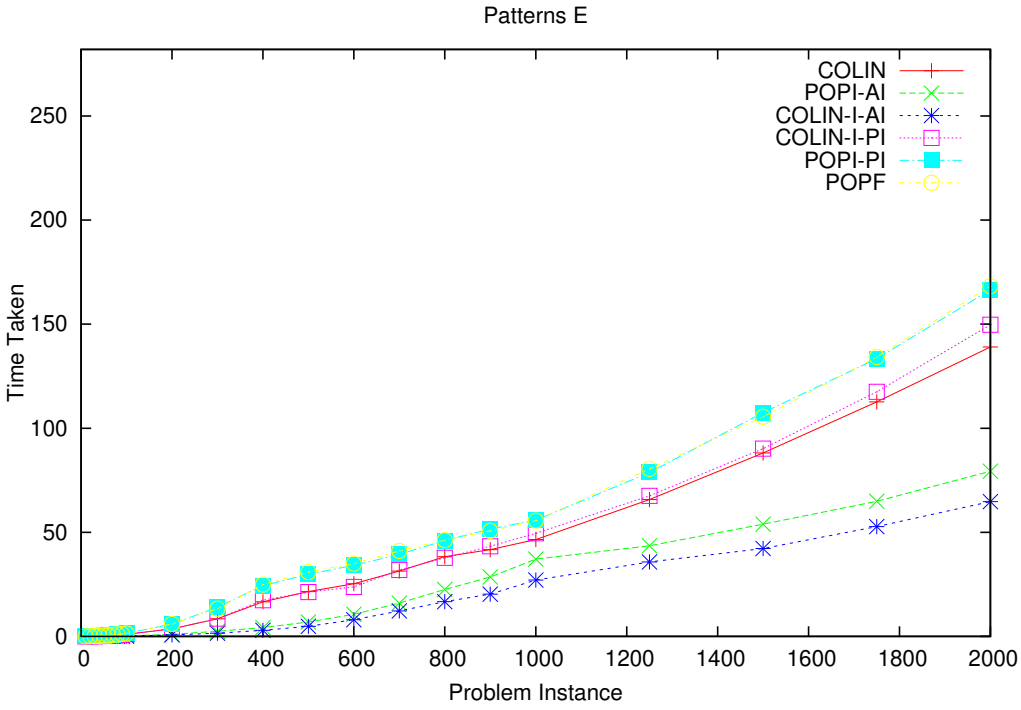



(b) Patterns D - Time Taken

Figure 7.5: Pattern D

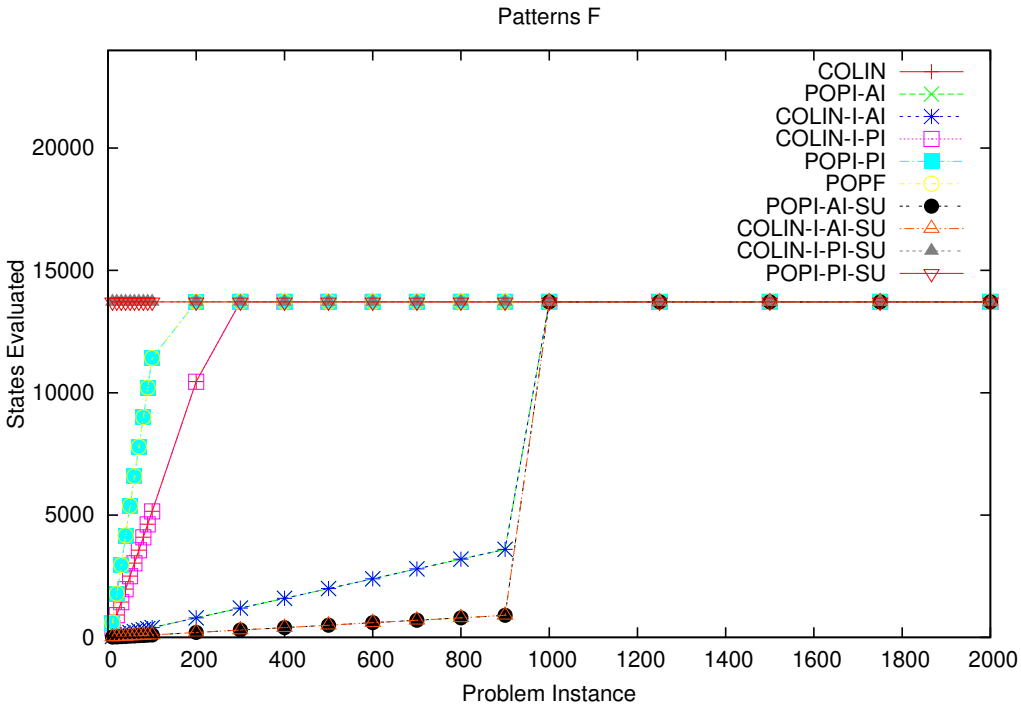



(a) Patterns E - States Evaluated

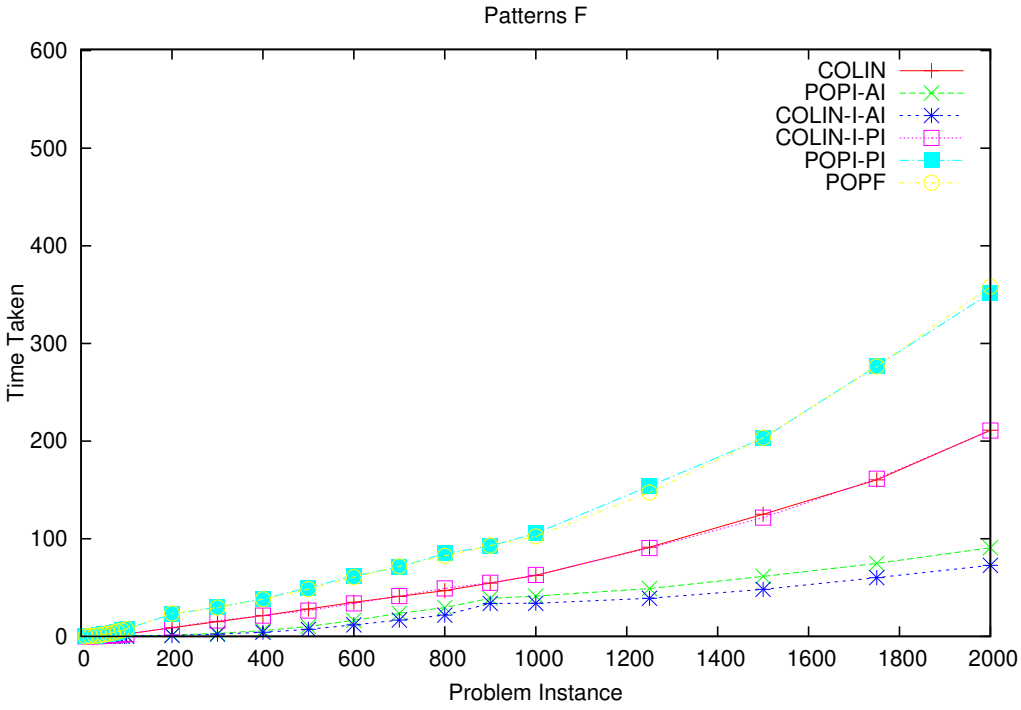



(b) Patterns E - Time Taken

Figure 7.6: Pattern E

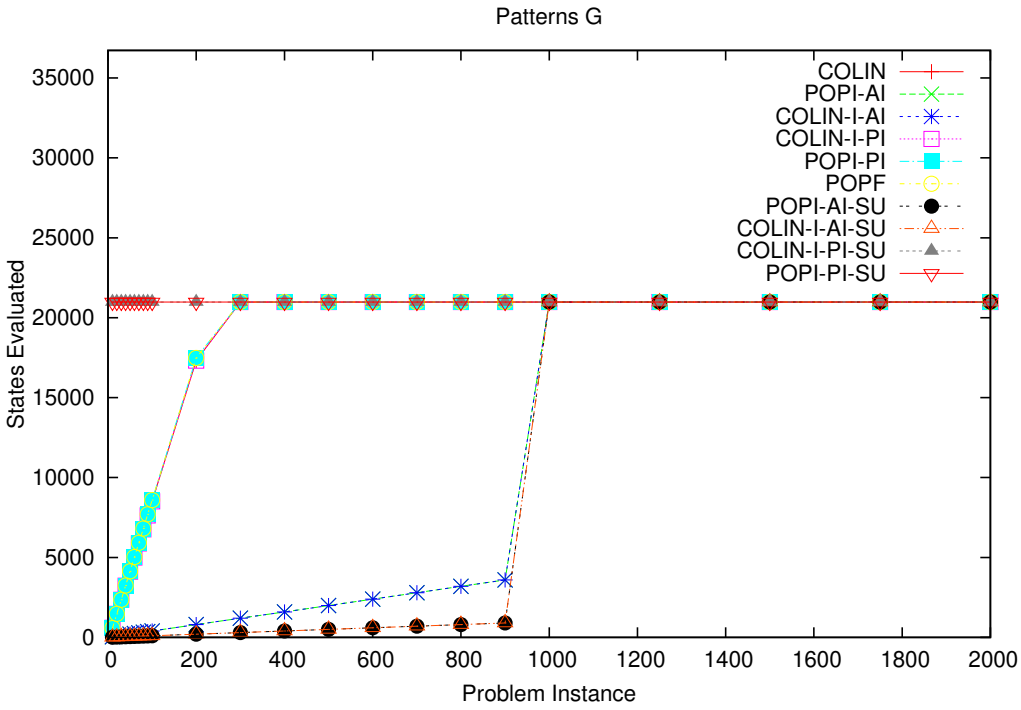



(a) Patterns F - States Evaluated

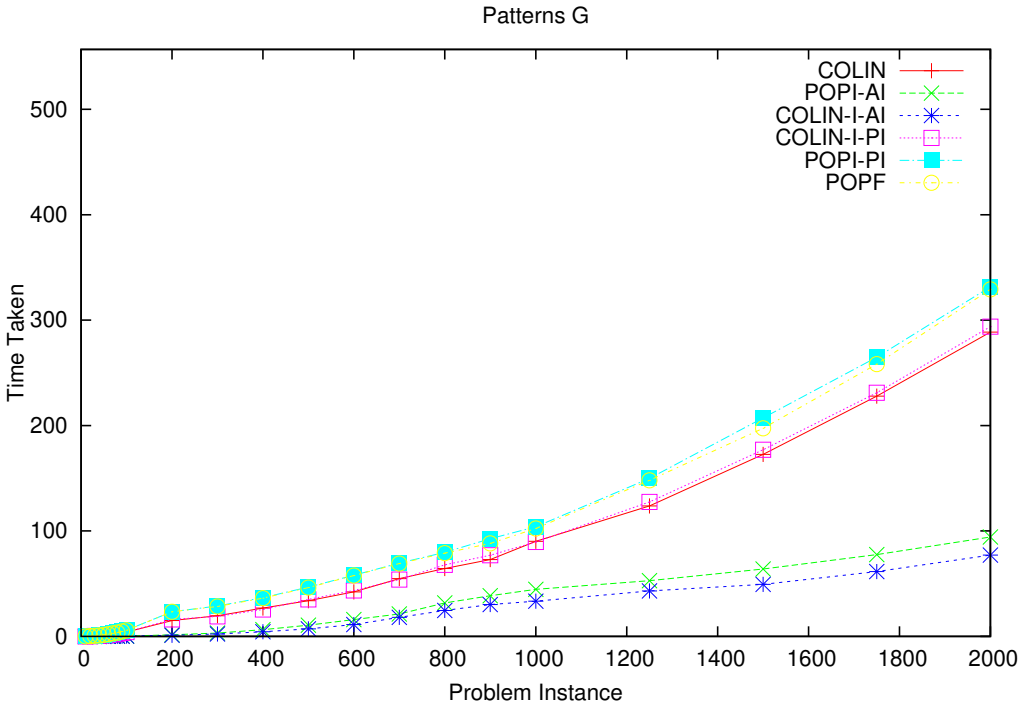



(b) Patterns F - Time Taken

Figure 7.7: Pattern F

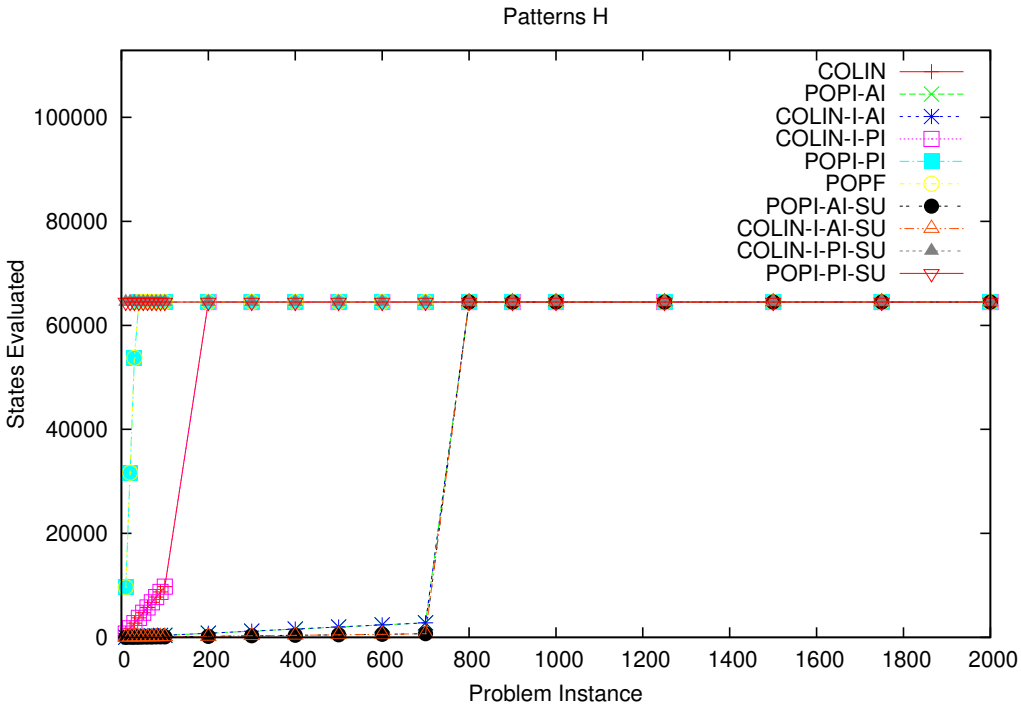



(a) Patterns G - States Evaluated

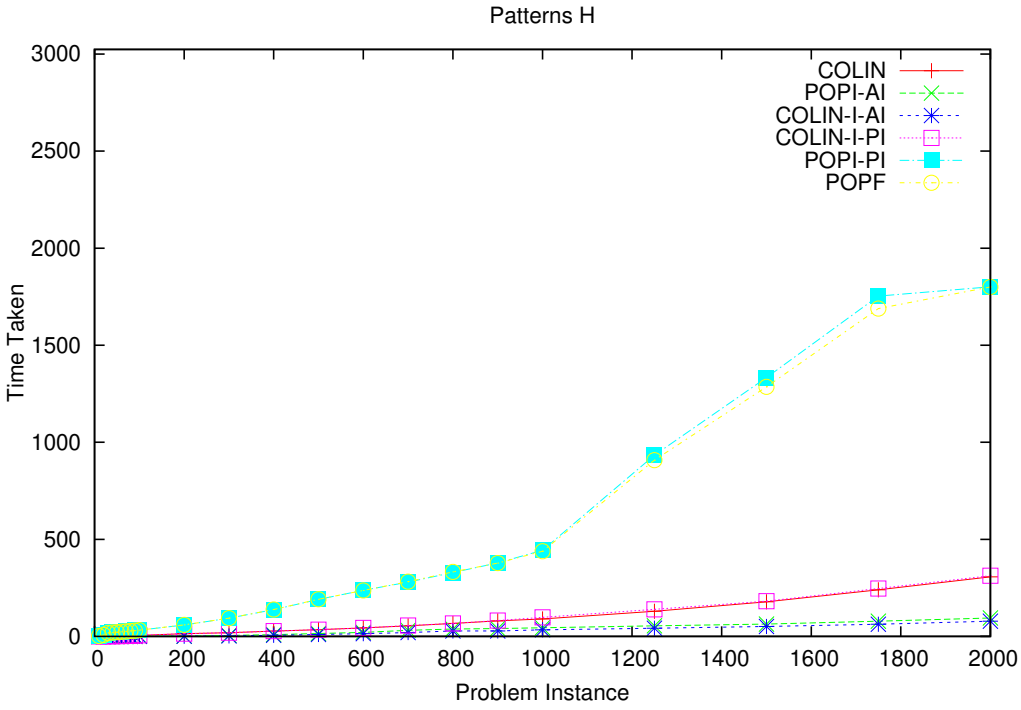



(b) Patterns G - Time Taken

Figure 7.8: Pattern G

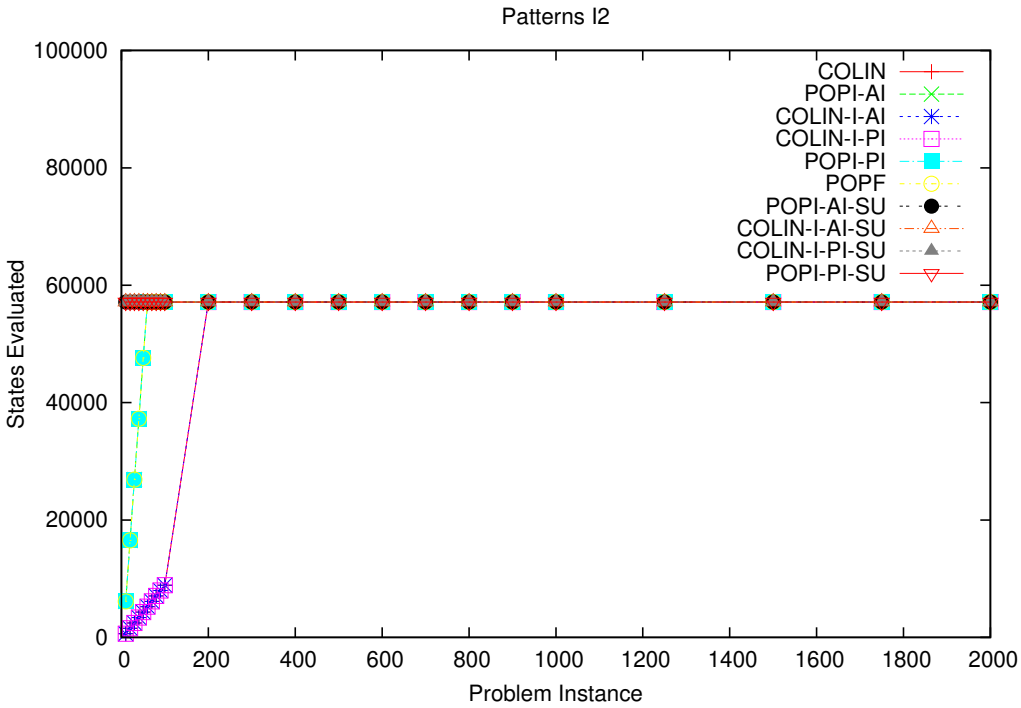



(a) Patterns H - States Evaluated

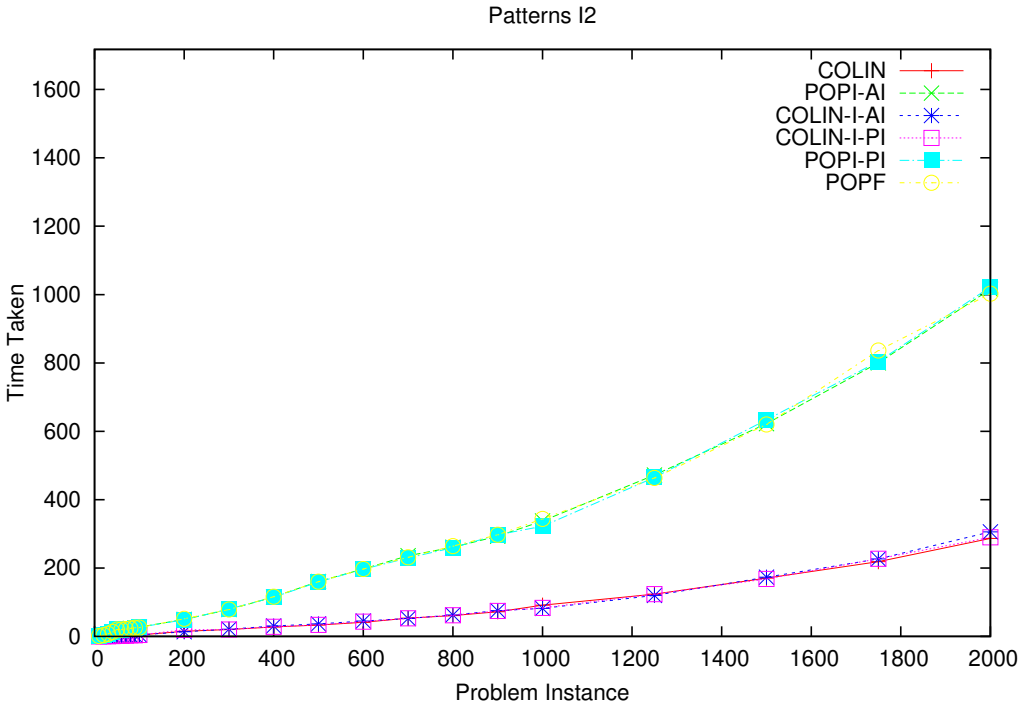



(b) Patterns H - Time Taken

Figure 7.9: Pattern H

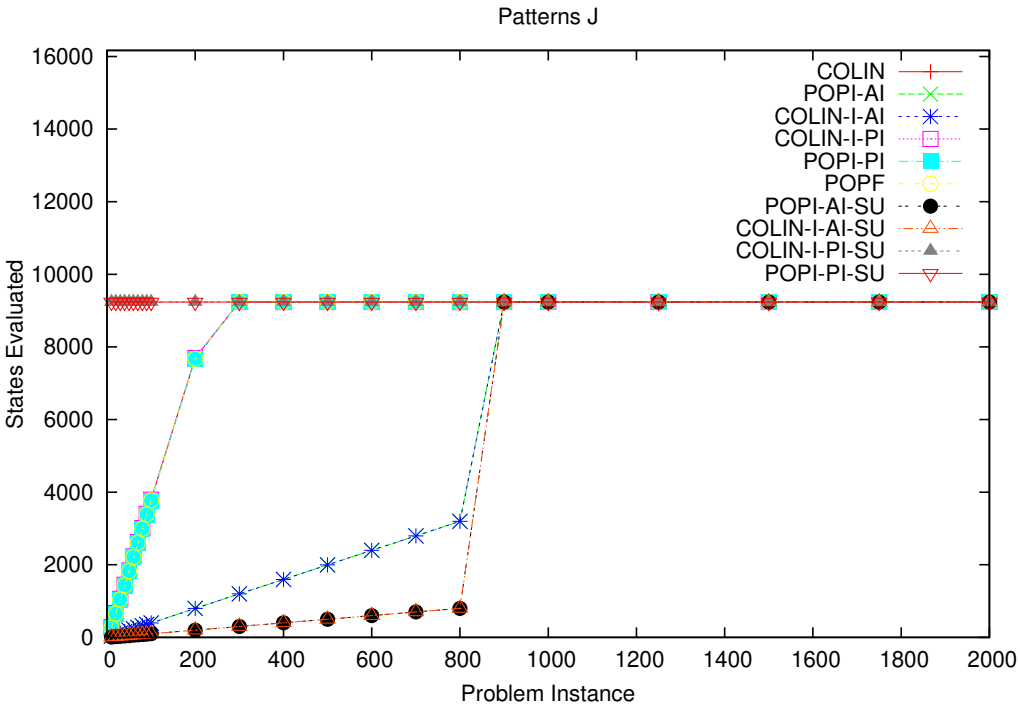



(a) Patterns I - States Evaluated

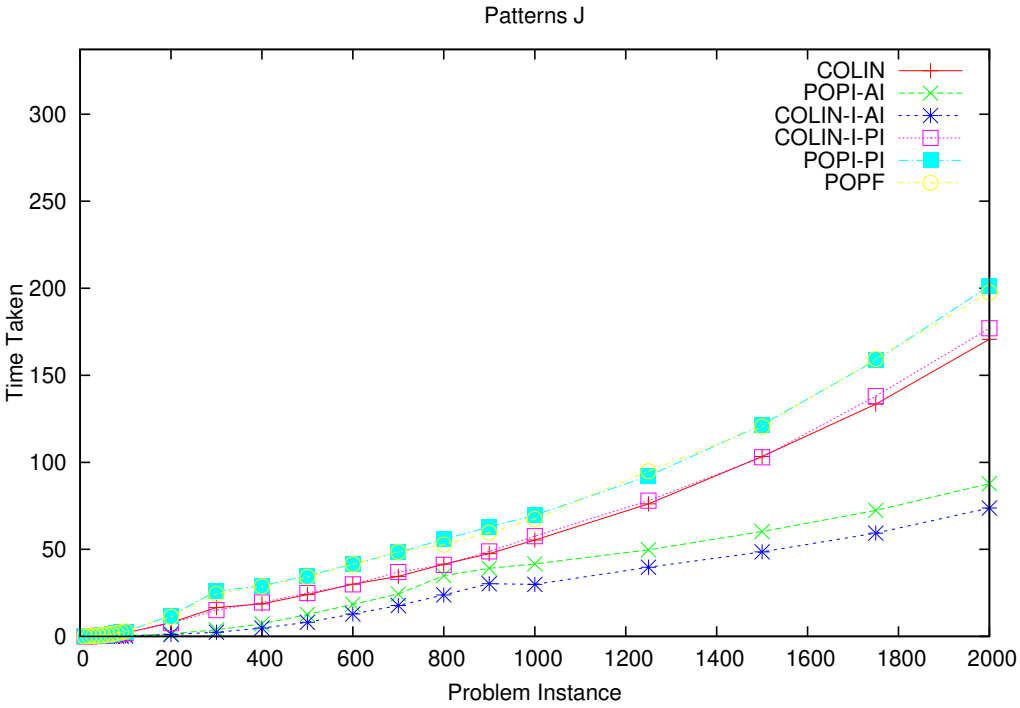



(b) Patterns I - Time Taken

Figure 7.10: Pattern I

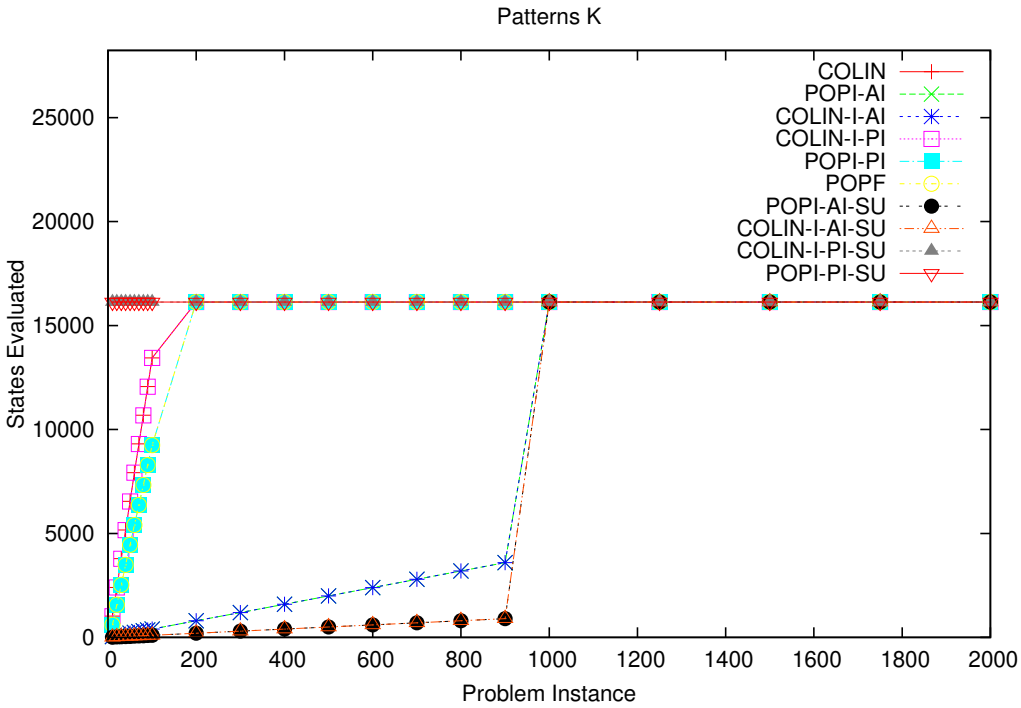



(a) Patterns J - States Evaluated

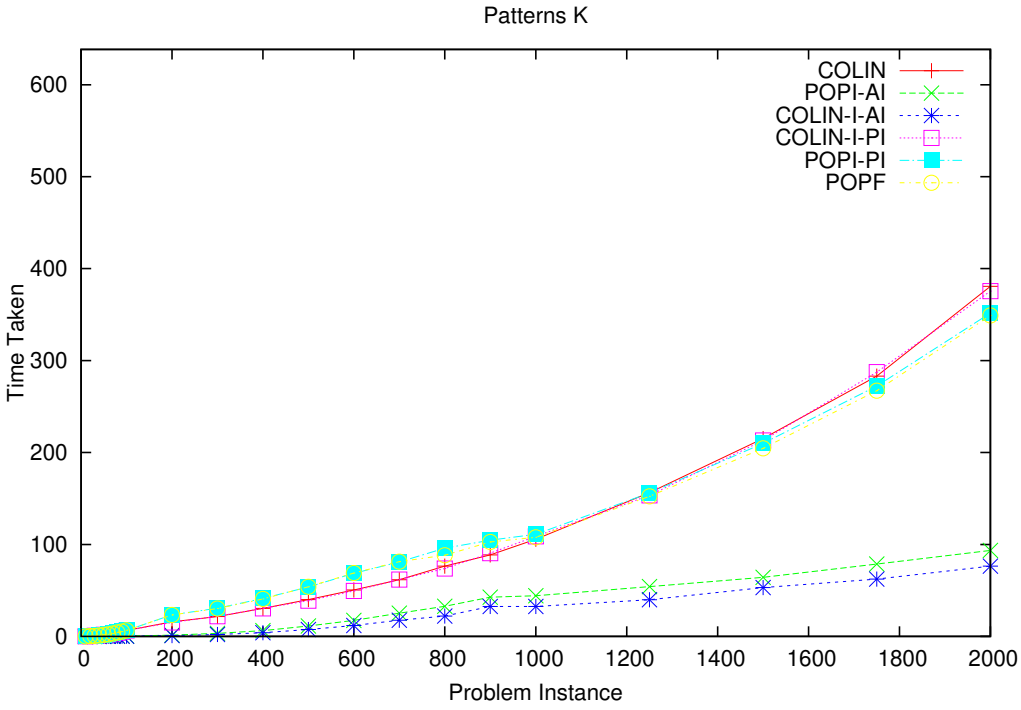



(b) Patterns J - Time Taken

Figure 7.11: Pattern J

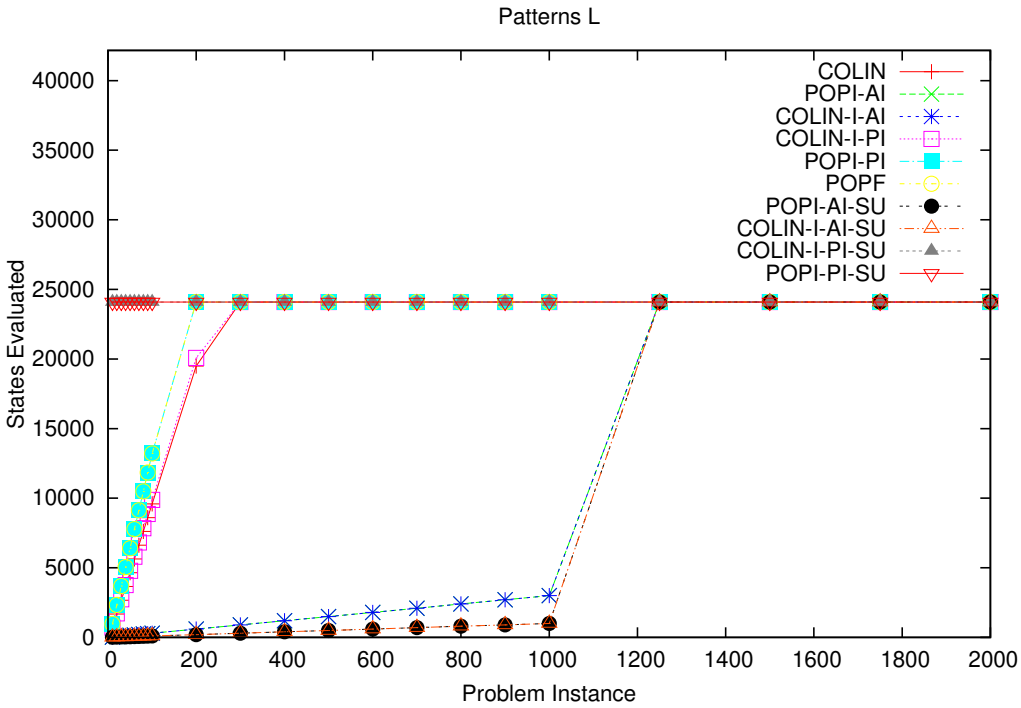



(a) Patterns K - States Evaluated

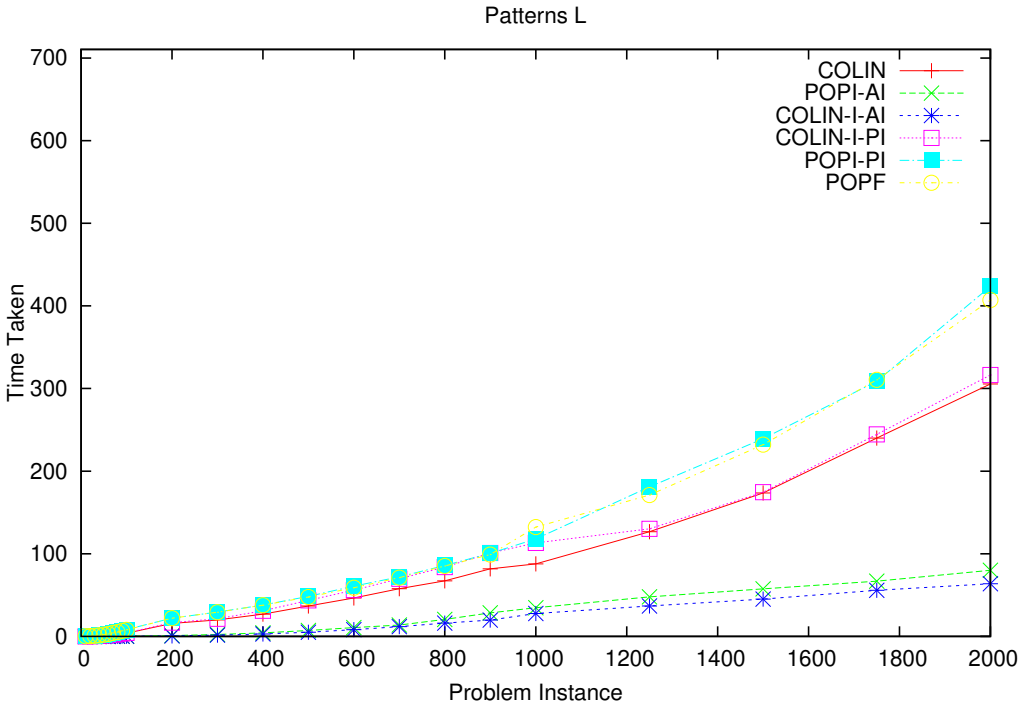



(b) Patterns K - Time Taken

Figure 7.12: Pattern K

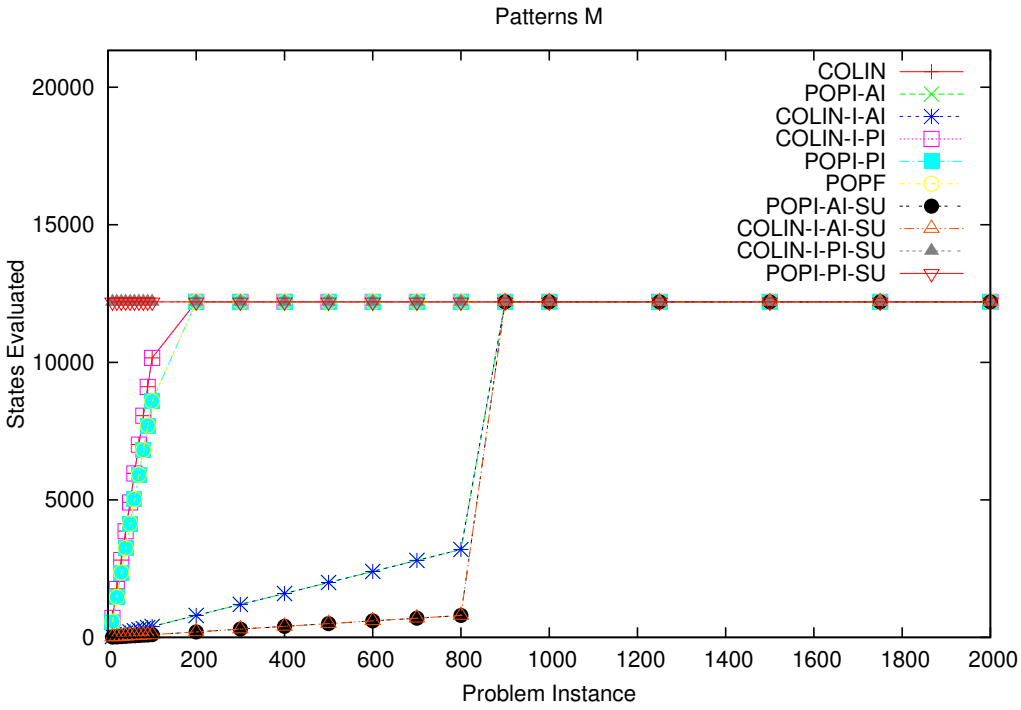



(a) Patterns L - States Evaluated

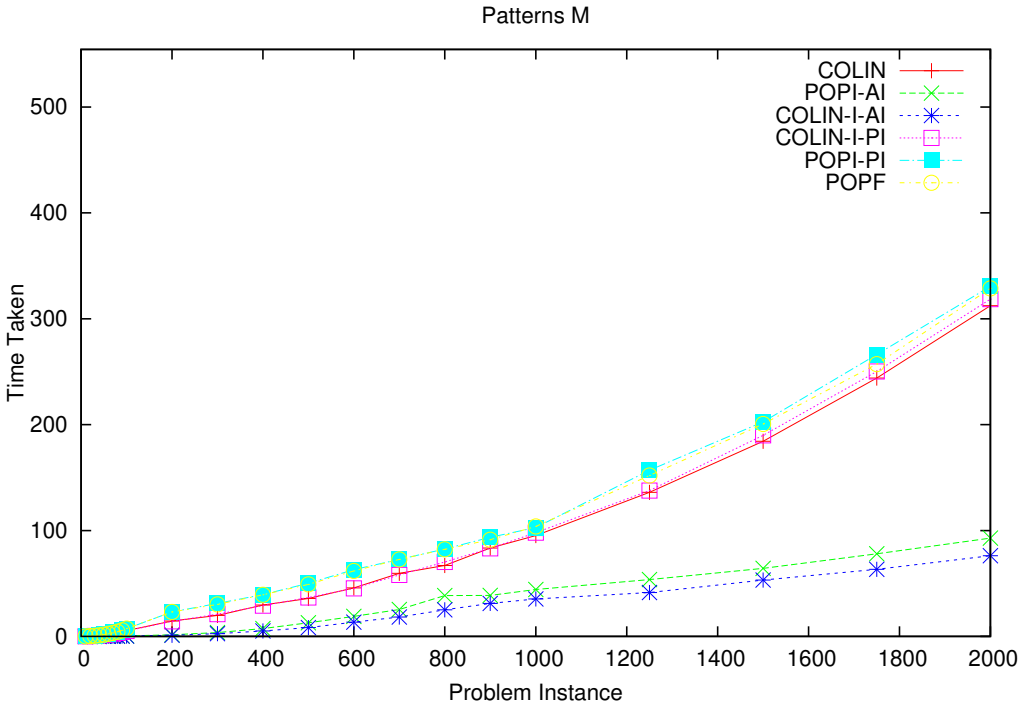



(b) Patterns L - Time Taken

Figure 7.13: Pattern L

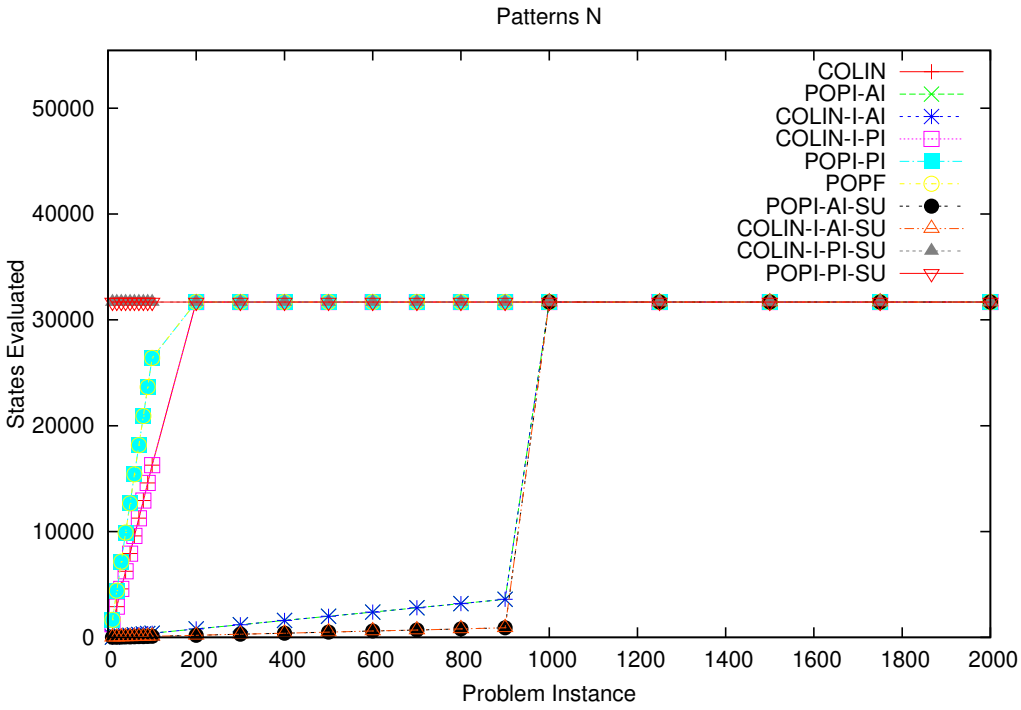



(a) Patterns M - States Evaluated

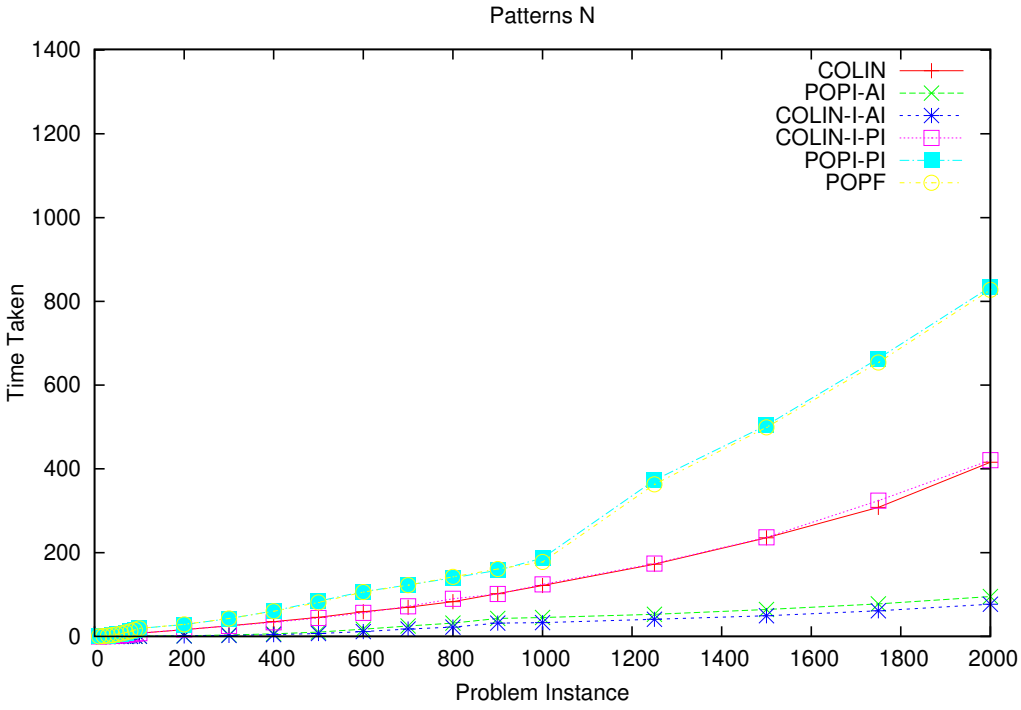



(b) Patterns M - Time Taken

Figure 7.14: Pattern M

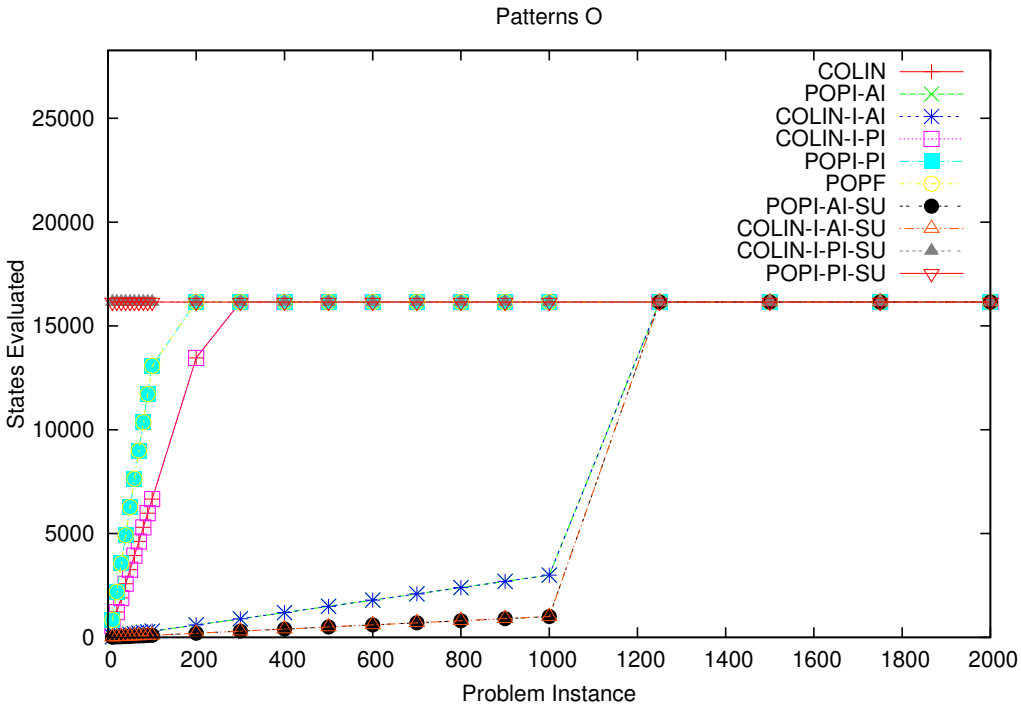



(a) Patterns N - States Evaluated

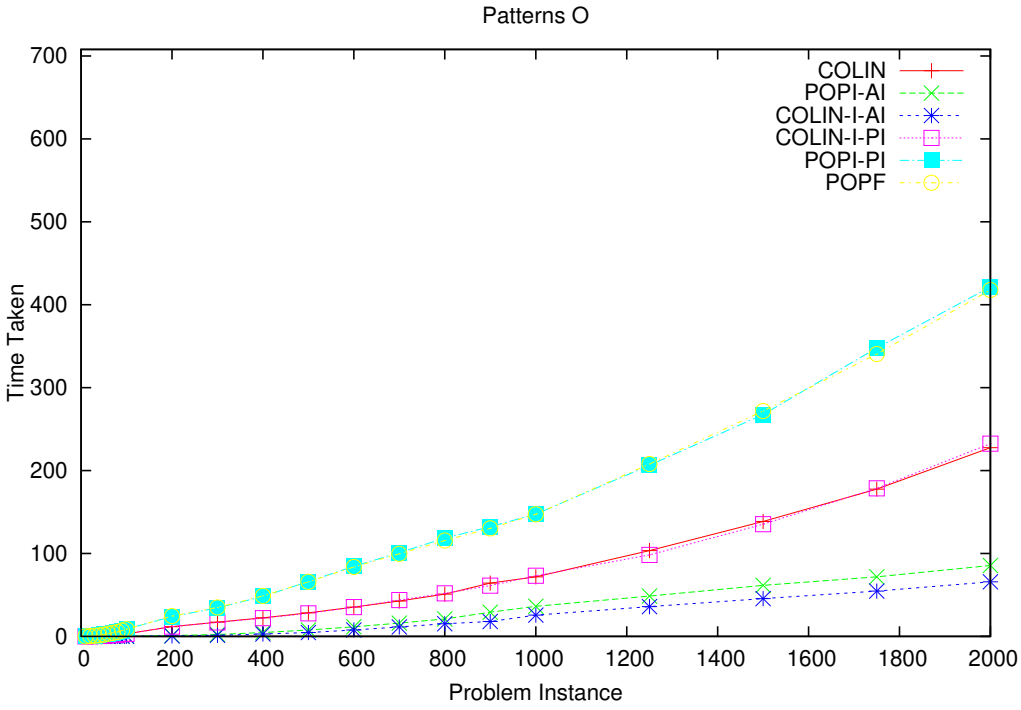



(b) Patterns N - Time Taken

Figure 7.15: Pattern N

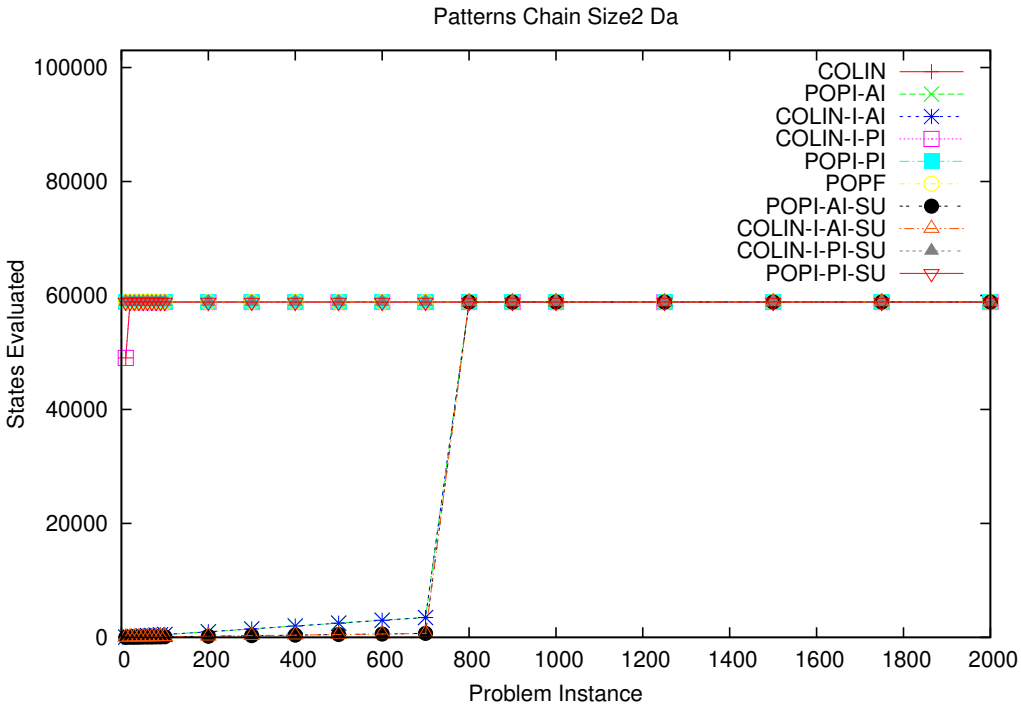



(a) Patterns O - States Evaluated

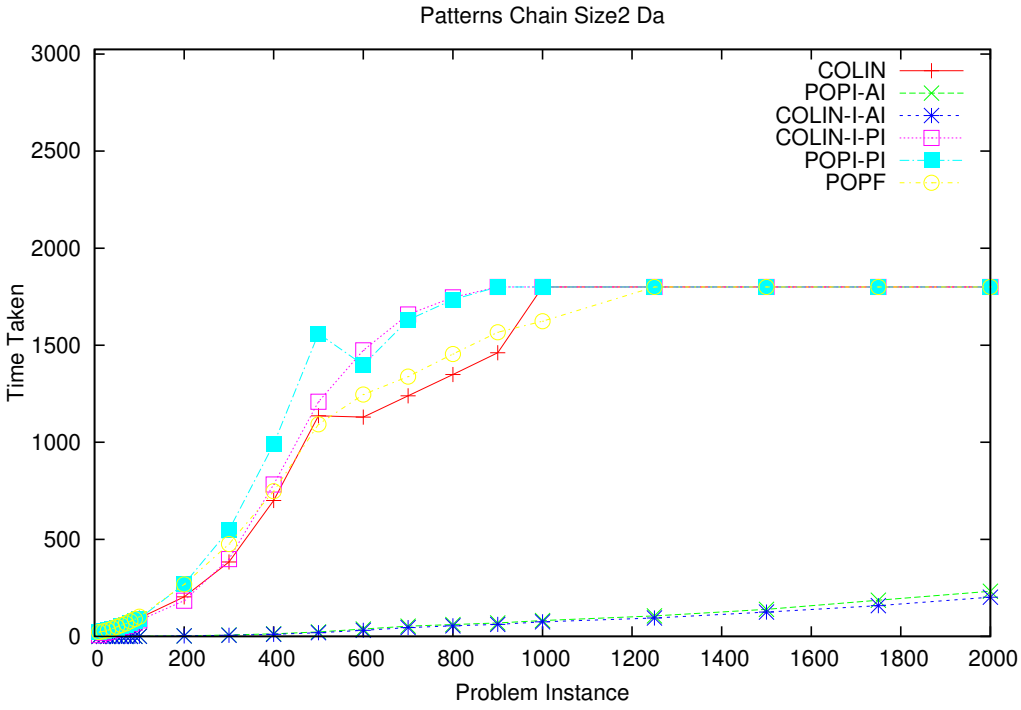



(b) Patterns O - Time Taken

Figure 7.16: Pattern O

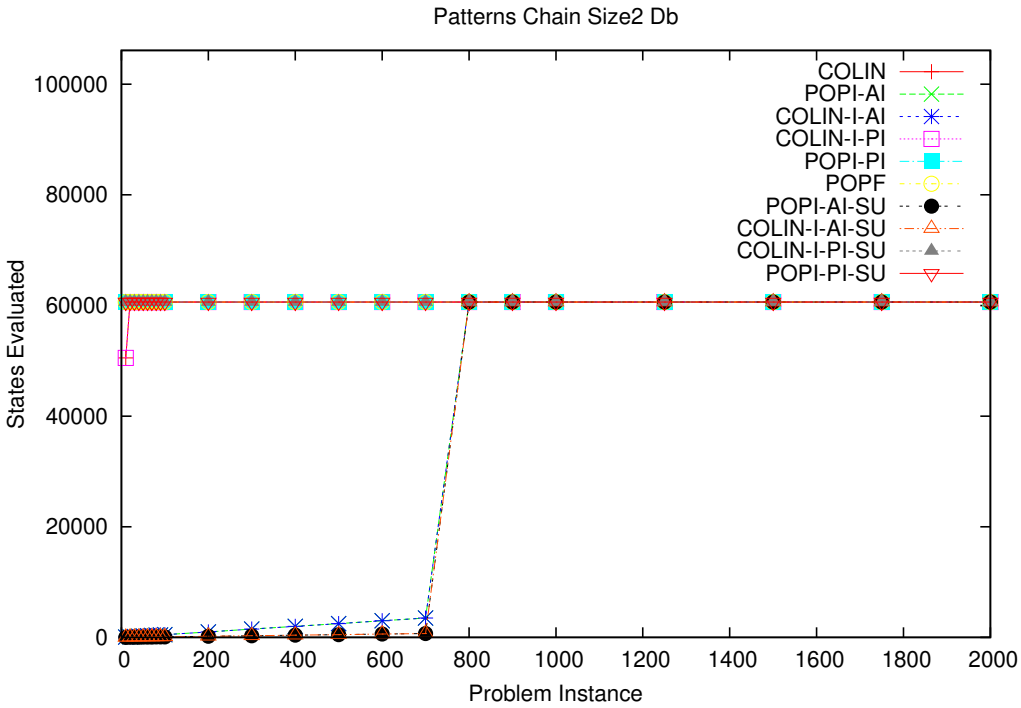



(a) States Evaluated

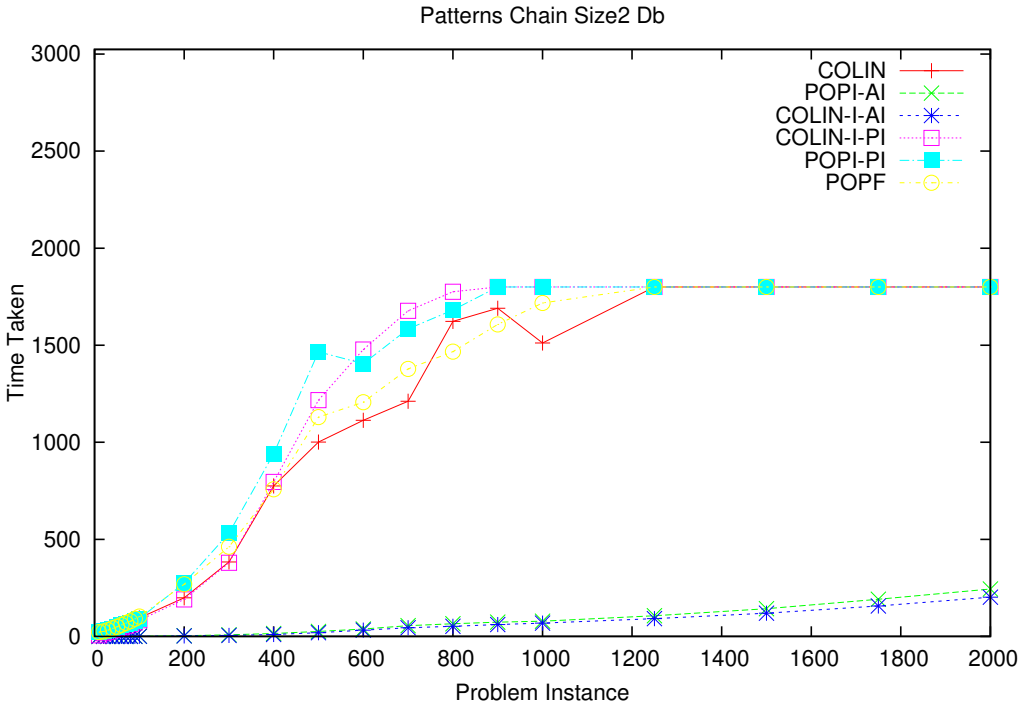



(b) Time Taken

Figure 7.17: Pattern Chain of size 2 containing types D and A

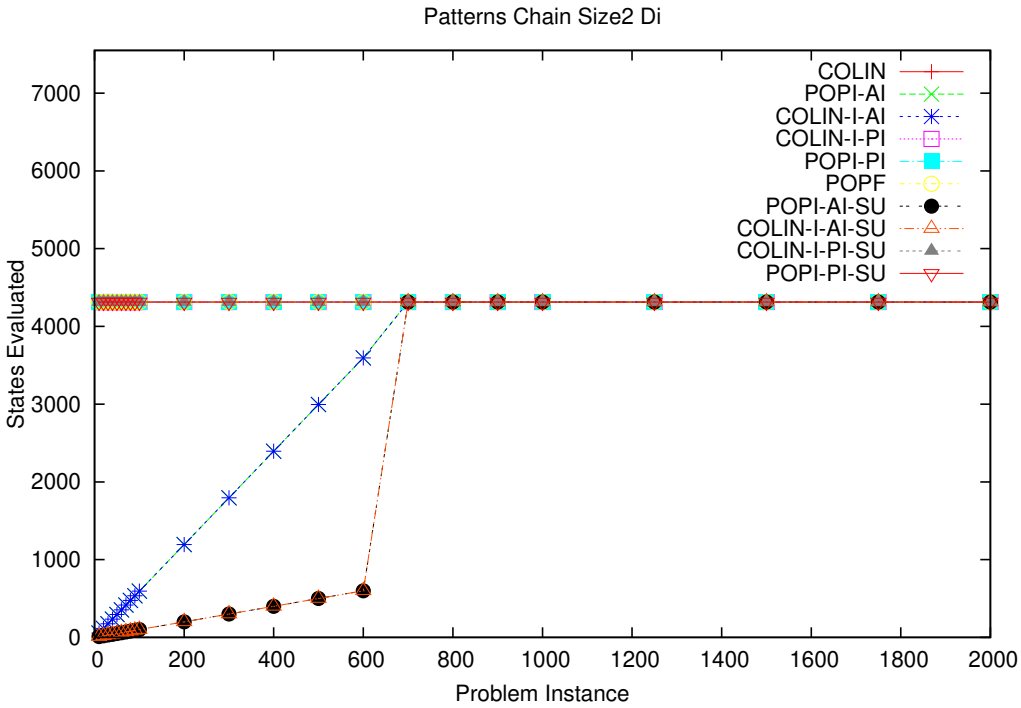



(a) States Evaluated

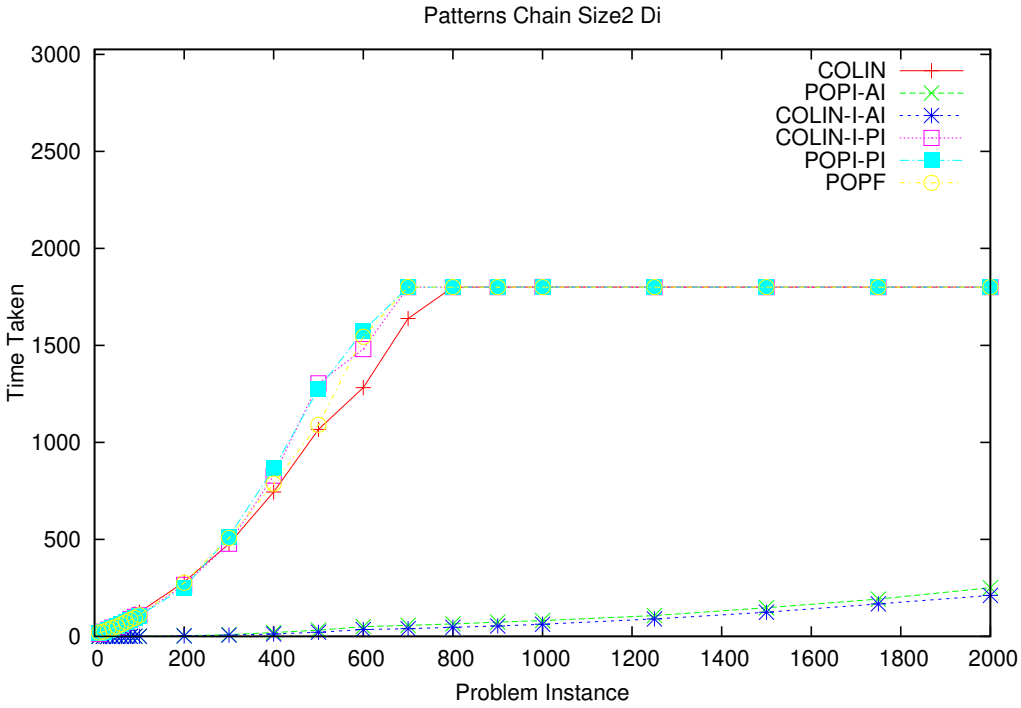



(b) Time Taken

Figure 7.18: Pattern Chain of size 2 containing types D and B

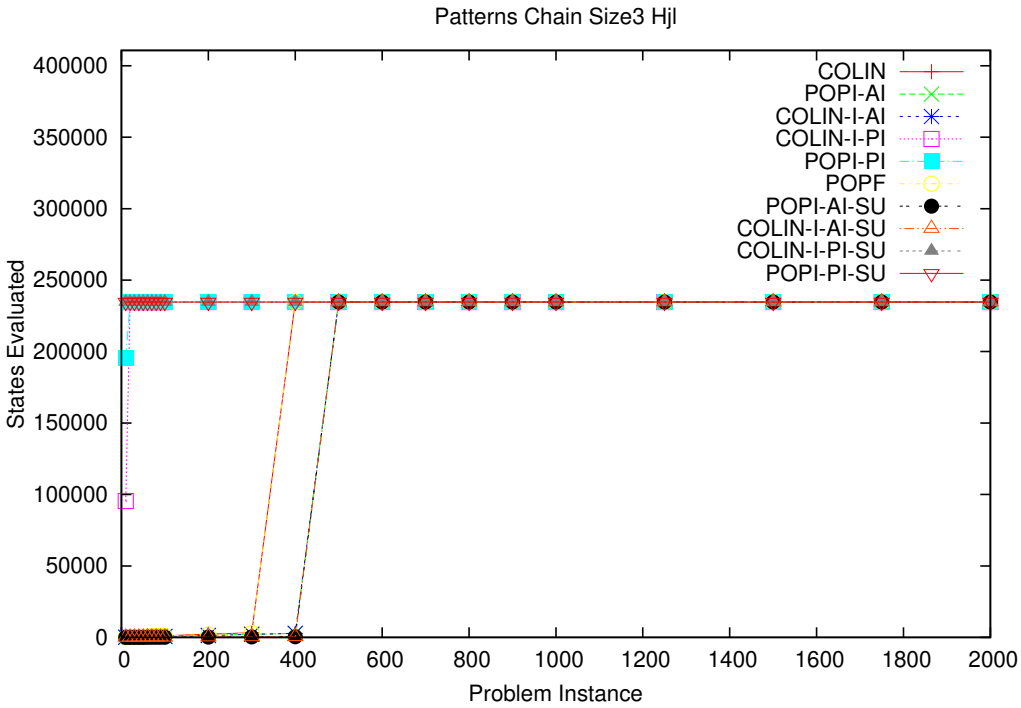



(a) States Evaluated

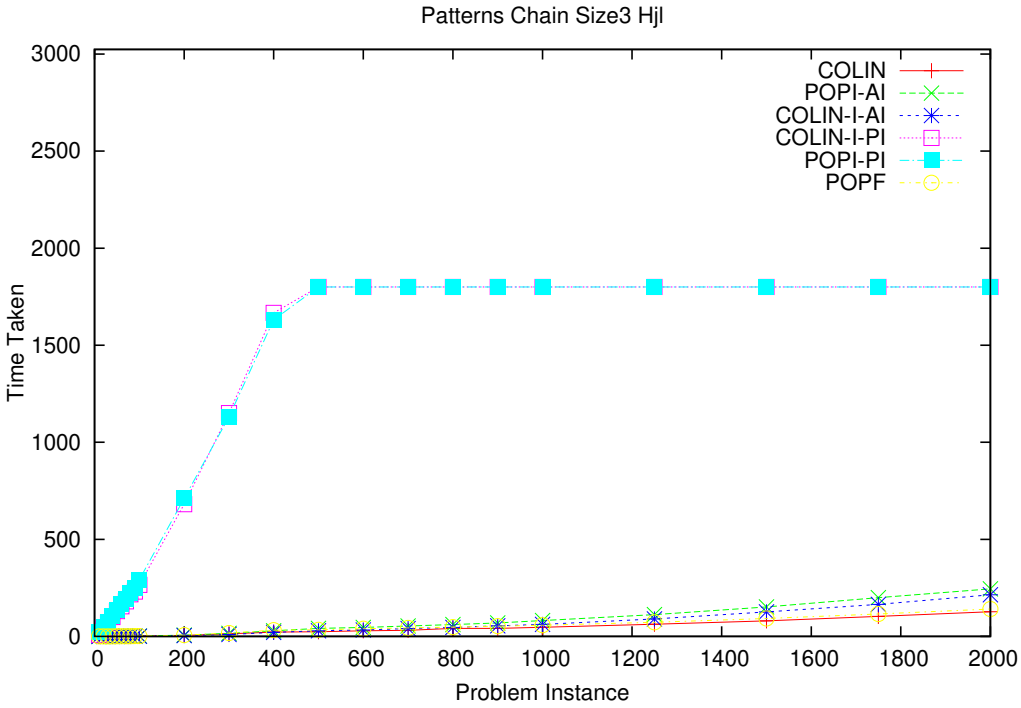



(b) Time Taken

Figure 7.19: Pattern Chain of size 2 containing types D and I

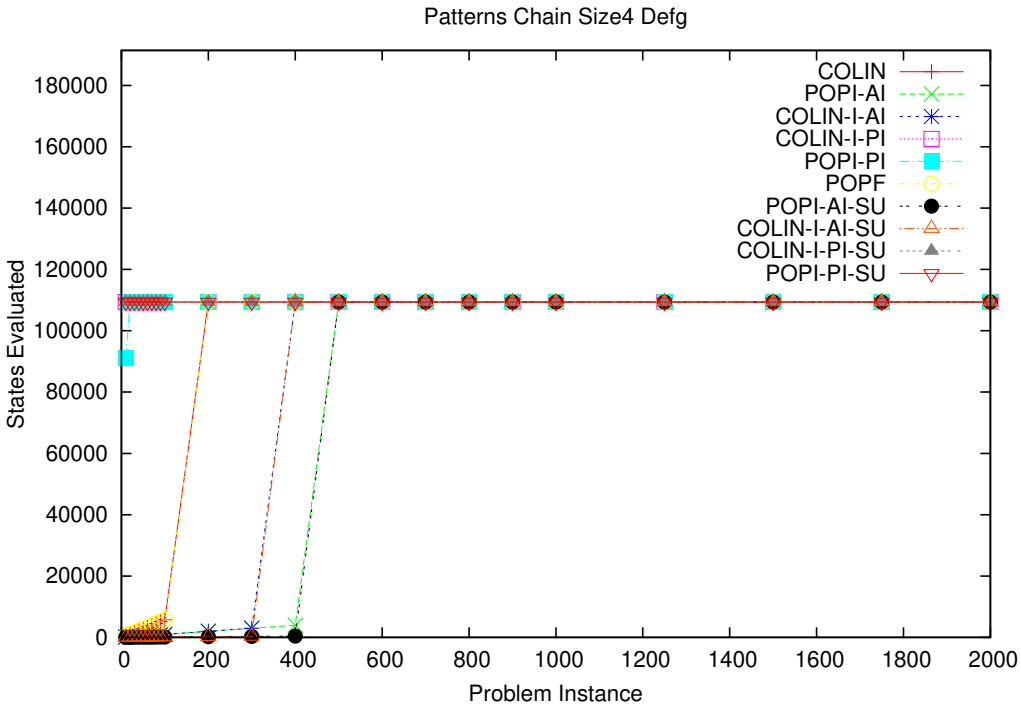



(a) States Evaluated

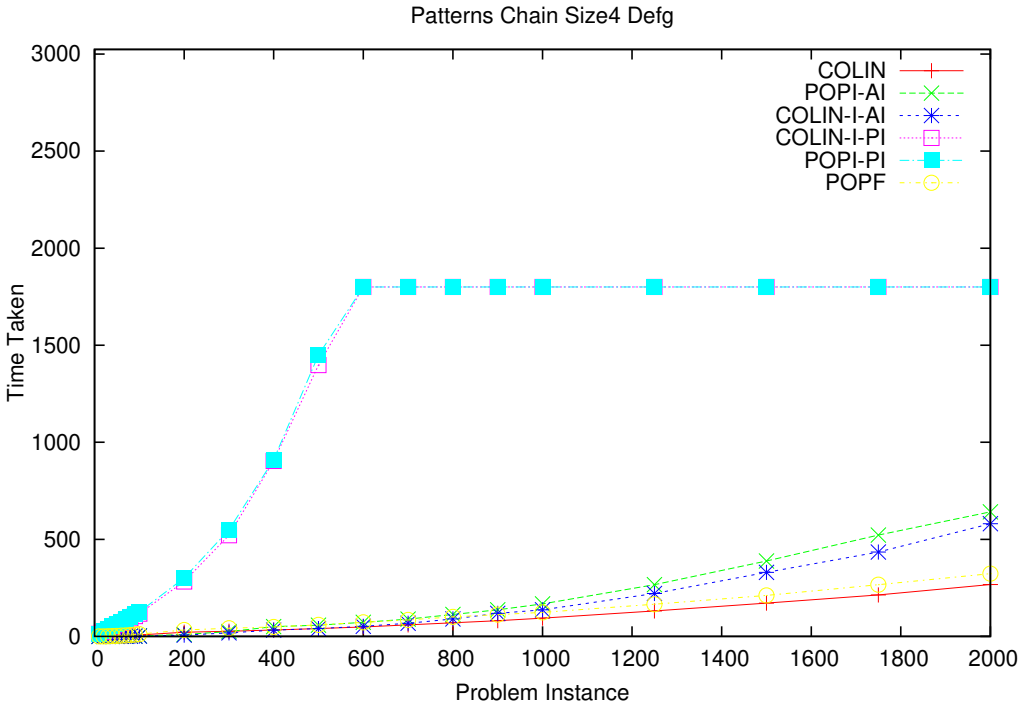



(b) Time Taken

Figure 7.20: Pattern Chain of size 3 containing types H, J and L

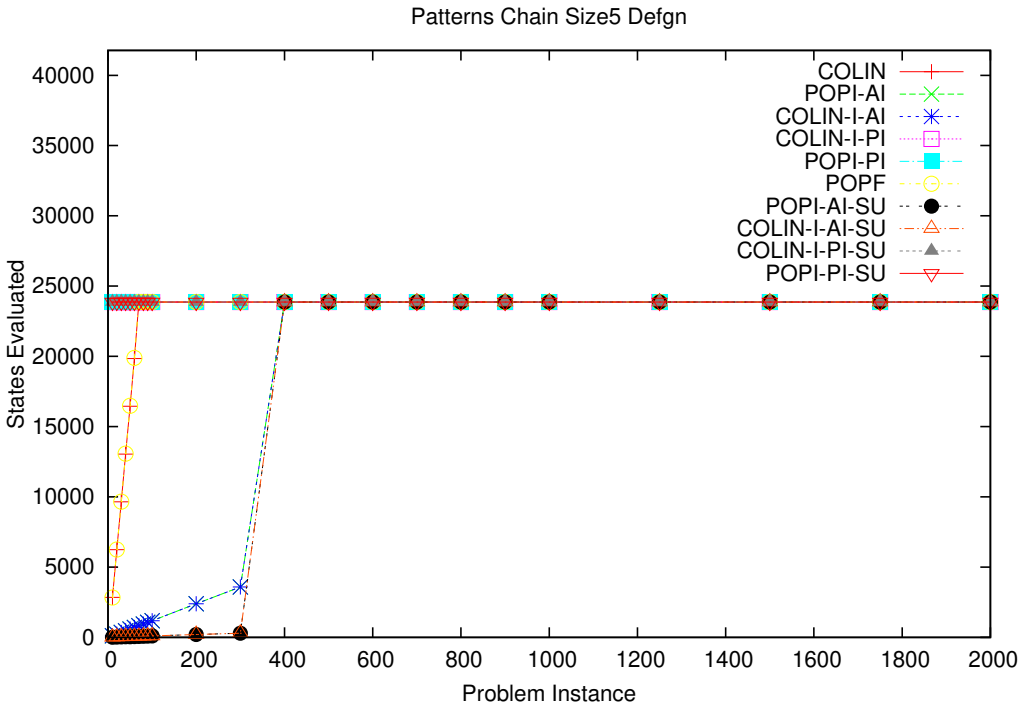



(a) States Evaluated

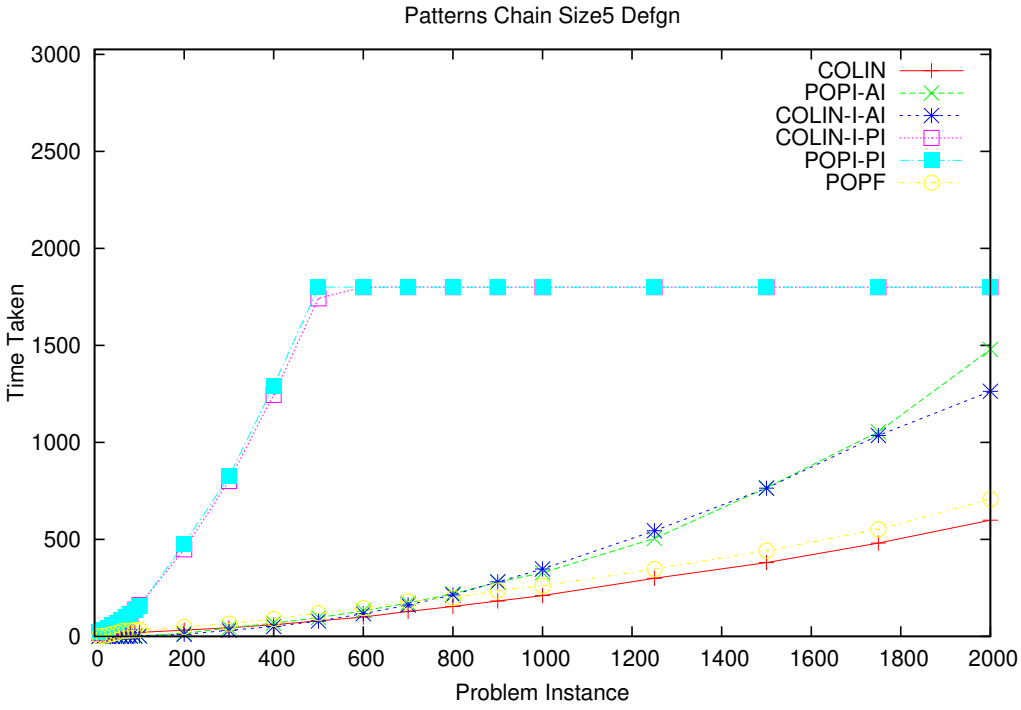



(b) Time Taken

Figure 7.21: Pattern Chain of size 4 containing types D, E, F, G




(a) States Evaluated




(b) Time Taken

Figure 7.22: Pattern Chain of size 5 containing types D, E, F, G, N

### 7.3.2 eCPT Experiments

In Chapter 2 we discussed CPT and its successor eCPT, which both use inference to solve problems. eCPT finds optimal plans by iteratively increasing the bound on the number of actions allowed in the plan, until it reaches a bound with the minimal number of actions needed to produce a plan. This allows eCPT to find optimal solutions. In this section we perform experiments using eCPT on the Patterns domains, as these domains contain the pattern structures handled within the scope of this thesis. We present the results of these experiments separately to those of POPI and its other comparisons, since eCPT and POPI work very differently and produce different measurements and output for their respective solutions. eCPT is not a state-based planner, so we cannot compare the results with POPI in terms of states evaluated. Therefore, the only useful comparison between POPI and eCPT is time taken to solve the problems for each of these domains. eCPT produces plans that have an optimal makespan, whereas the plans produced by POPI are not guaranteed to be optimal. POPI simply looks to solve the problem. However, in both the case of eCPT and POPI, they produce their respective plans as the first and only solution when solving a problem, which allows us to compare the time taken to solve each problem.

The 15 Patterns domains each with a single pattern of every type have been modified to have fixed duration actions, as eCPT does not handle duration inequalities. The smallest problem that we used for experiments in the Pattern domains was with 10 objects, where nine applications of the action pair in the pattern need to be applied to achieve the goal. However, eCPT did not manage to find a plan within the 30 minutes time limit for the experiments. For this reason we created smaller problem instances to run for eCPT, such that we could observe its behaviour in the context of these domains with patterns of required concurrency. The problems instances with object sizes of 2, 5, 8, 9 and 10 require 1, 4, 7, 8 and 9 applications of the pattern to be applied respectively, in order to achieve the goal for these problems. We do not go beyond 10 objects, since this problem is not solved by eCPT for any of the Pattern domains.

#### Results

Figure 7.23 presents the time taken to solve the five problems of each of the Patterns domains. Observing that none of the size 10 problems are solved by eCPT shows that although some the smaller problems are solved quickly, eCPT clearly does not scale well in comparison to POPI using its aggressive or even its passive strategy or indeed POPF. These results show us that eCPT consistently reached the 30 minutes (1800 seconds) time limit set for all the experiments. Although eCPT could not solve many of these problems, it has shown itself to be memory efficient. POPI using its aggressive inference strategy solved problems on a much larger scale, but consumed memory at a faster rate.

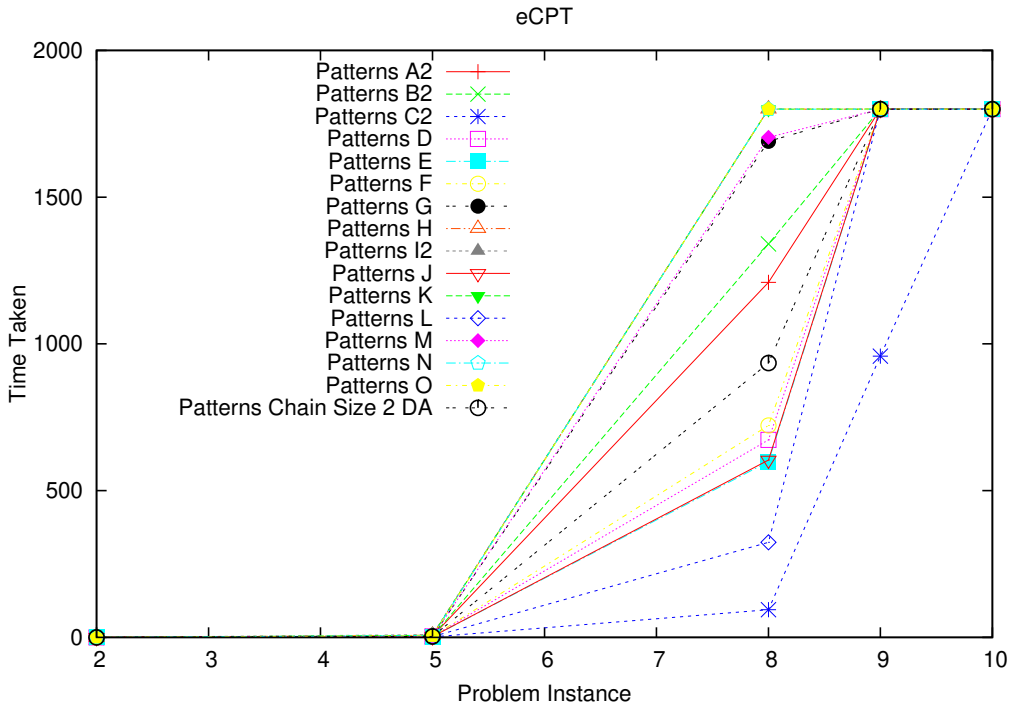



Figure 7.23: Time Taken to solve problems.

**Discussion**

All problems that were tested with each version of the `patterns` domain are solved using mostly inference and only a small amount of search with POPI's aggressive strategy. The reason for this is that starting at the initial state in each problem, there is an applicable action that triggers the pattern. POPI prioritises selection of this action instead of standard helpful actions, and treats trigger actions as being the most helpful of all. Each application of the inferred actions in the pattern results in POPI-AI and its total ordering version, COLIN-I-AI getting closer to the goal, at which point another applicable action of the same pattern structure becomes applicable and is applied in the same fashion and the same process is repeated until the goal is reached. In contrast to this, POPF becomes lost in a large search space due to existence of a single extra action at each state, that the planner always attempts to apply this action first, since the heuristic incorrectly informs the POPF that this action will take it closer to the goal. The constraint based planner eCPT is able to solve small size problems from the Patterns domains, but struggles to scale to even the smallest problem tested with POPF and the POPI planners.

### 7.3.3 TFD Experiments

Temporal Fast Downward is a well known temporal planner that we ran tests on using the Patterns domains with fixed durations, as was the case for eCPT. This is because TFD is

also limited to handling fixed duration actions. We start the tests by running the base case problem with 2 objects with each domain. After performing initial tests, it is clear that TFD does not produce consistently correct plans in amongst the 15 single Patterns domains, where each one contains a pattern structure of each type. For this reason it did not make sense to run larger scale experiments for TFD in using these domains.

**Results of Tests**

The clear error in the output for the test runs using TFD in these domains, was that for 14 of the 15 patterns domains, each plan produced consisted of only one action. This was action $A$ and no action $B$ component was included in the plan, even though it is needed to successfully solve the problem. The only exception to this was for the pattern I domain test, where $A$ and $B$ were both in the plan, however, the plan was temporally invalid and not scheduled correctly. All of the produced plans were run on a Validator tool with their domain and problem files and it confirmed that they were indeed invalid plans. The reason that TFD fails on these domains is because action $A$, which is the one that achieves the goal, has an end precondition and TFD ignores this precondition as it cannot schedule the start of action $B$ relative to the start of action $A$. Figure 7.24a shows the action pair in the pattern type F domain used for testing. Figure 7.24b shows the plan produced by TFD and we can see that this is incorrect. Although action $A$ achieves the goal, it needs $B$ to make fact $q$ true which is its end precondition.

In order to determine whether TFD could correctly solve a problem for a domain with a pattern structure in any form, we used another version of the pattern A domain, which was altered to have no end preconditions, even ones external to the pattern structure. Although end preconditions are not part of the structure for pattern type A, there was one for the action $A$ in the first version of the pattern A domain, used for the POPI experiments, that was an external precondition. Figure 7.25a shows the domain extract for a pair of actions in a pattern type A structure, where TFD produces a valid plan. We can see that in this pattern A structure, there is no end precondition for action $A$, and the end precondition of action $B$ being ignored does not matter in this case, since `(ready ?a)` is already true when the end of $B$ is applied. Figure 7.25b shows the plan produced for this pattern A domain problem where the plan is valid.

**Summary**

Having tested TFD on the patterns domains, it is clear that it is not able to solve any problem in most of these domains, since the planner ignores end preconditions, and many of the pattern types have structures with end preconditions as part of them. For pattern A, we have seen an example of a problem that it is able to solve by movingthe application of $B$ start

```
(:durative-action Act_A
 :parameters(?a ?b - typeA )
 :duration(= ?duration 5)
 :condition(and (at start (active)) (at end (q ?a))
  (at start(next ?a ?b)) (at start (ready ?a)) (at end(ready ?a)) )
 :effect (and (at start(p ?a)) (at start(not(active))) (at start(ready ?b))
    (at end (active)) (at end(not(ready ?a)))
   )
 )

(:durative-action Act_B
 :parameters(?a  - typeA)
 :duration(= ?duration 4.5)
 :condition(and (at end(p ?a))  (at start (ready ?a)))
 :effect (and (at end (q ?a)) )
)
```

(a) Pattern F action pair.

```
0.00100000:  (act_a obj1 obj2) [5.00000000]
```

(b) TFD plan for Pattern F domain problem.

Figure 7.24: Actions from Pattern F domain where TFD produces an invalid plan.

```
(:durative-action Act_A
 :parameters(?a  - typeA)
 :duration(= ?duration 5)
 :condition(and (at start (ready ?a)))
 :effect (and (at start(p ?a)) (at end(not(p ?a))))
)

(:durative-action Act_B
 :parameters(?a ?b - typeA )
 :duration(= ?duration 4.5)
 :condition(and (at start (active)) (at start(next ?a ?b)) (over all(p ?a))
  (at start (ready ?a)) (at end(ready ?a)) )
 :effect (and (at start(not(active))) (at start(ready ?b)) (at end (active))
    (at end(not(ready ?a)))
   )
 )
```

(a) Pattern A action pair.

```
0.00100000:  (act_a obj1 obj2) [5.00000000]
0.01100000:  (act_b obj1 obj2) [4.50000000]
```

(b) TFD plan for Pattern A domain problem.

Figure 7.25: Actions from Pattern A domain where TFD produces a correct plan.

to immediately after $A$ start, ignoring that fact that $A$ deletes the invariant condition of $B$ at its end. Since action $A$ ends after the end of $B$, the problem is correctly solved.

### 7.3.4 IPC Temporal Planners

In order to test the competitiveness of POPI on problems of required concurrency, we now test other state-of-the-art temporal planning systems that competed in the temporal satisficing track of the International Planning Competition 2018. We test these planners on the simplest problem for each of our patterns domains to determine which patterns can be handled. TFD which was also component planner of TemPorAl portfolio entered into IPC 2018 was tested in Section 7.3.3. The TFD results for the Patterns domains were presented separately, since there were solutions outputted with invalid plans for which we presented examples and analysis of problems solved correctly and incorrectly.

#### Results of Tests

Table 7.1 summaries which of the additional temporal planners that we have tested are able to solve the simplest problem for each of the Patterns domains, containing each of the pattern types that POPI has been designed to handle. Each of the Patterns domains are defined with duration inequalties. This is important since many of the pattern structures depend on their being flexible durations, because otherwise the choices in the orderings of the happenings that should exist for certain patterns, do not exist with fixed durations. Pattern G is an example of this, where the pattern structure is such that all four of the valid application orderings of the happenings can occur. This is only possible if the planner can decide the duration of actions A and B, where either one can be longer than the other. As we can see in Table 7.1 none of the planners tested here are able to solve problems from our Patterns domains with flexible duration actions. TPSHE, TP($K$), for $K \in \{2, 3, 4\}$ and STP($K$), for $K \in \{2, 3, 4\}$ are able to solve problems of pattern type A when defined with fixed durations, but TFLAP and ITSAT cannot. TPSHE is limited to handling fixed duration actions and for problems of required concurrency, only handles problems containing single hard envelopes, so the results it produced for these tests were as expected. TP(K) and STP(K) are also limited to handling fixed duration actions, therefore the results of the tests for patterns A to O are not surprising on these planners. The pattern F test problem with fixed durations was not solved using any of the planners in Table 7.1.

#### Summary

It is clear that these temporal planners entered into the IPC temporal satisficing track are not able to solve problems from the different Patterns domains that each contain a pattern structure. In temporal planning, problems of required concurrency are more complex to solve

| | TPSHE | TP(2) | TP(3) | TP(4) | STP(2) | STP(3) | STP(4) | TFLAP | ITSAT |
|---|---|---|---|---|---|---|---|---|---|
| A | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| B | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| C | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| D | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| E | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| F | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| G | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| H | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| I | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| J | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| K | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| L | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| M | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| N | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| O | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| A($F$) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| F($F$) | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |

Table 7.1: Pattern types handled by IPC temporal planners. The first column shows the pattern type for each row. Domain for each pattern type is defined with duration inequalities, with additional tests for fixed duration versions of pattern A and F in the final two rows appended with ($F$).

compared to problems that allow sequential solutions. Required concurrency problems can become more complex when there are duration inequalities involved, where the planner must decide the durations of actions as in the case of the Patterns domains. Since the planners tested in Table 7.1 did not solve problems with our pattern structures on the smallest problems requiring one application of the two actions in each pattern, we did not test them on any larger problems. In the two fixed duration patterns domains, for patterns A and F, none of the IPC planners tested solved the pattern F test problem. Most of the planners were only able to solve the fixed duration pattern A test problem, however TFLAP and ITSAT did not produce solutions for the pattern A test either. It is possible that these planners could solve problems with fixed durations for some of the other pattern domains, however we are primarily interested pattern domains with duration inequalilties. Overall, the results and analyses presented in this section satisfy the second evaluation objective listed at the beginning of this chapter.

## 7.4 Optional Concurrency Domains with Patterns

The Patterns domains were designed to present a set of cases illustrating a type of scenario where POPI using its aggressive inference strategy has the opportunity to showcase its capabilities in performance as a result of the pattern based inference it performs. This is both in terms of solving problems with less state evaluations and scaling better to solve larger size problems that other planning strategies cannot manage. In this section we analyse results from some of the Patterns domains, which have been altered, such that in this version, the third action outside of the pattern can be used to reach a solution. The application of the pattern actions is optional, hence these are domains with optional concurrency. We examine the difference in performance and trends over problem size for the POPI, POPF, COLIN and COLIN-I planners, but do not test the other planners from Section 7.3.4 since it was clear that they were not able to solve problems in the various patterns domains with duration inequalities.

Figure 7.26 shows the actions from the alternate version of the Patterns G domain with the pattern type G structure that still exist between actions `Act_A` and `Act_B`. The difference is that now `Act_C` can be used to achieve the problem goal in this version of the domain compared with the first patterns G domain experimented with in Section 7.3.

```
(:durative-action Act_A
 :parameters(?a ?b - typeA )
 :duration(and (<= ?duration 5) (>= ?duration 1))
 :condition(and (at start (active)) (at end (q ?a))
  (at start(next ?a ?b))  (at start (ready ?a))  (at end(ready ?a)) )
 :effect (and (at start(p ?a)) (at start(not(active)))  (at start(ready ?b))
    (at end (active)) (at end(not(ready ?a)))
   )
 )

(:durative-action Act_B
 :parameters(?a  - typeA)
 :duration(and (<= ?duration 5) (>= ?duration 1))
 :condition(and (at end(p ?a))  (at start (ready ?a)))
 :effect (and (at start (q ?a))   )
)

  (:durative-action Act_C
 :parameters(?a ?b - typeA)
 :duration(and(<= ?duration 0.9 ) (>= ?duration 0.5))
 :condition(and (at start(ready ?a)) (at start(next ?a ?b)) (over all (ready ?b)))
 :effect (and (at start(not(ready ?a))) (at start(ready ?b)))
)
```

Figure 7.26: Actions from alternate version of Patterns G domain where Act_C can be used to achieve the goal.

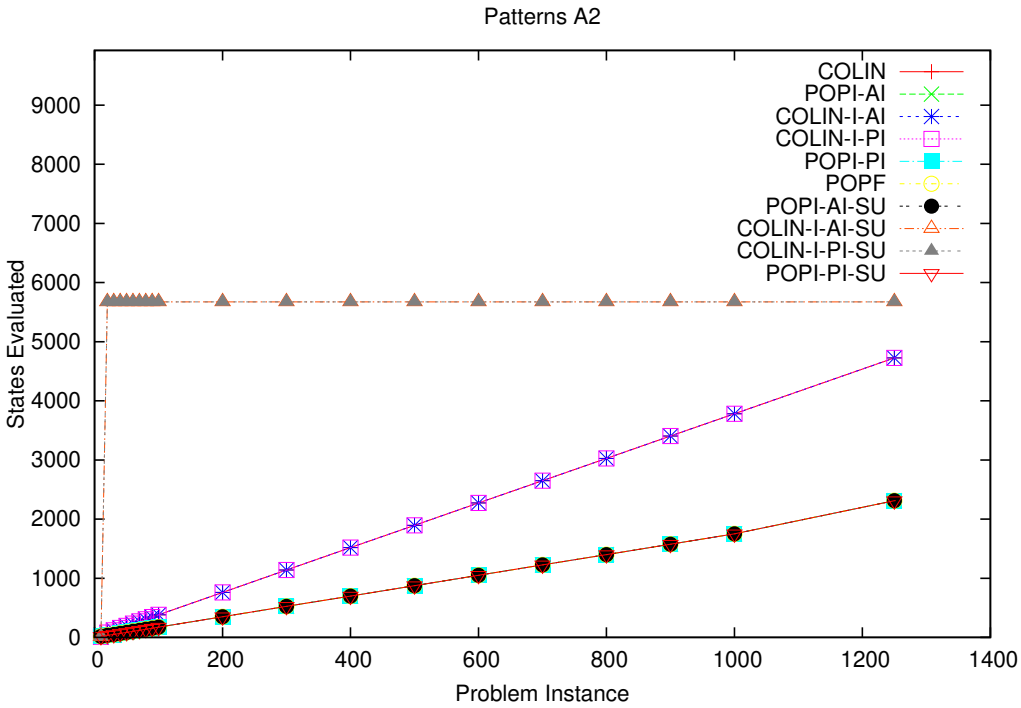
### 7.4.1 Results

In this set of domains we are interested in seeing how the relative performance of POPI-AI differs to POPF. We test a subset of the patterns domains including patterns A, G and M and two chained pattern domains, one with a chain of two patterns and the other with a chain of three pattern instances. For these experiments, we used problem instances with a size of up to 1250 objects as this is as larger problem as any of the planners could solve, within the time and memory limits given. All of these domains show the performance of POPI-AI to be similar to its performance in the counterpart domain with required concurrency presented in the previous section. The exception to this, which was expected, is for the optional concurrency pattern A domain results presented in Figure 7.27, where we see that POPI and POPF both do better in solving problems in terms of states evaluated. This was expected, because the use of the actions in the pattern are optional, and there is a third action that can be used to achieve the goal, meaning that the problems are easier to solve. Hence, problems of a much larger size are also solved, as well as being solved with far fewer state evaluations. We also see that for this pattern A domain, the problems are also solved much quicker.

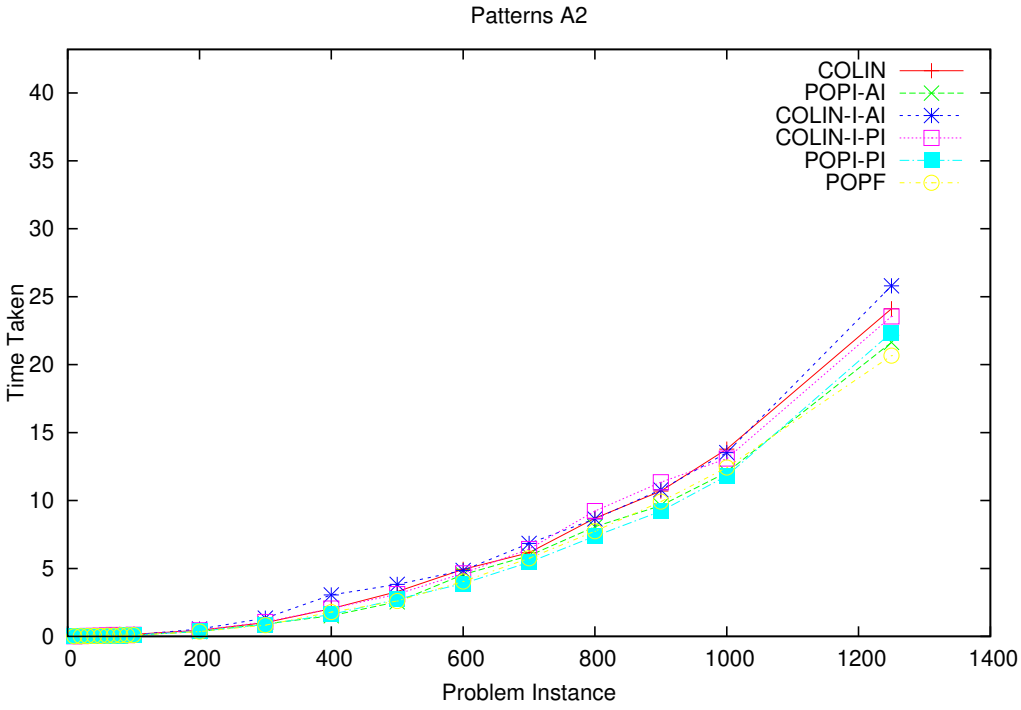
For the optional concurrency pattern G, presented in Figure 7.28, we see that the number of problems solved by POPI-AI is the same as the previous version of this domain, but now POPF is more competitive and actually solves more problems than POPI-AI. This suggests that POPF uses the sequential solutions to solve problems, whereas POPI-AI still aggressively pursues application of a pattern of actions where it is available. However, for the problems that both of these planners solve, we can see in Sub-figure 7.28a that POPI-AI solves them with fewer state evaluations used, than POPF's states evaluated. This is again the case for the optional concurrency version of the pattern M domain problems shown in Figure 7.29a. This is also the same behaviour in the two optional concurrency chained pattern domains of chain sizes 2 and 3 shown in Figures 7.30a and 7.31 respectively. In all of the optional concurrency patterns domains, the amount of information gain decreases compared to their counterpart domains in Section 7.3.1. We can see that the relative performance between POPI-AI and POPF comes closer together, with POPF solving more problems in the optional concurrency domains. This was with exception of the pattern A domain because POPI-AI does search in the same manner as POPF, since it cannot infer any new action in the pattern A domain.

### 7.4.2 Summary

We can see from the results presented in this section, that when the application of actions in a pattern structure are optional, the performance of POPF improves and comes closer to POPI-AI, however POPI-AI still solves many problems in less state evaluations for domains where it can infer new actions that take it closer to the goal.

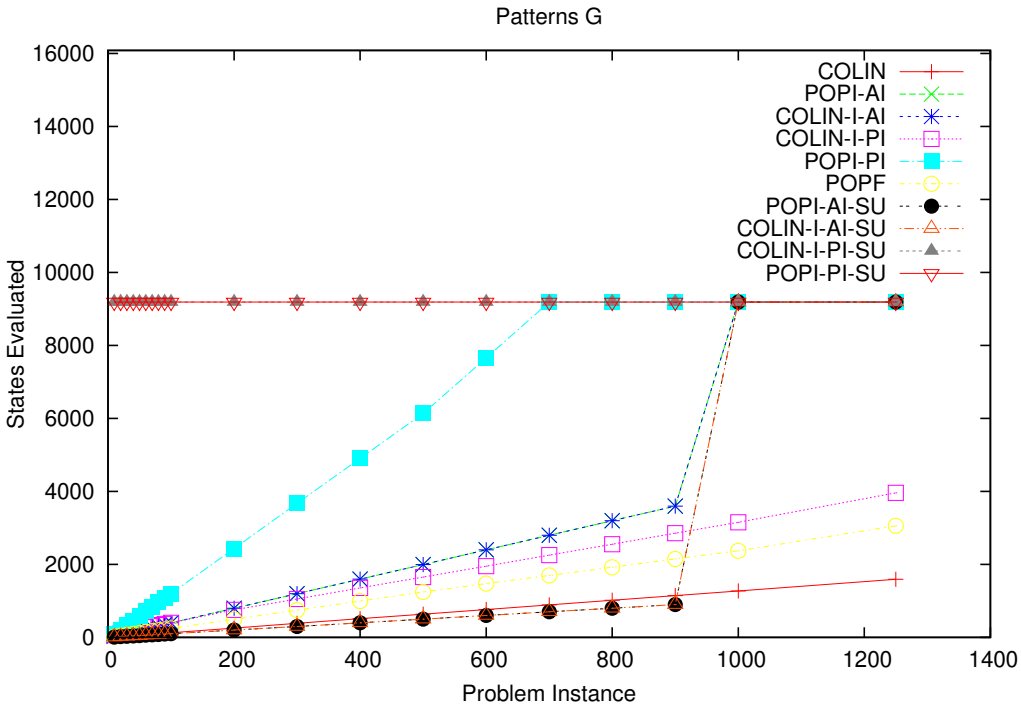



(a) States Evaluated

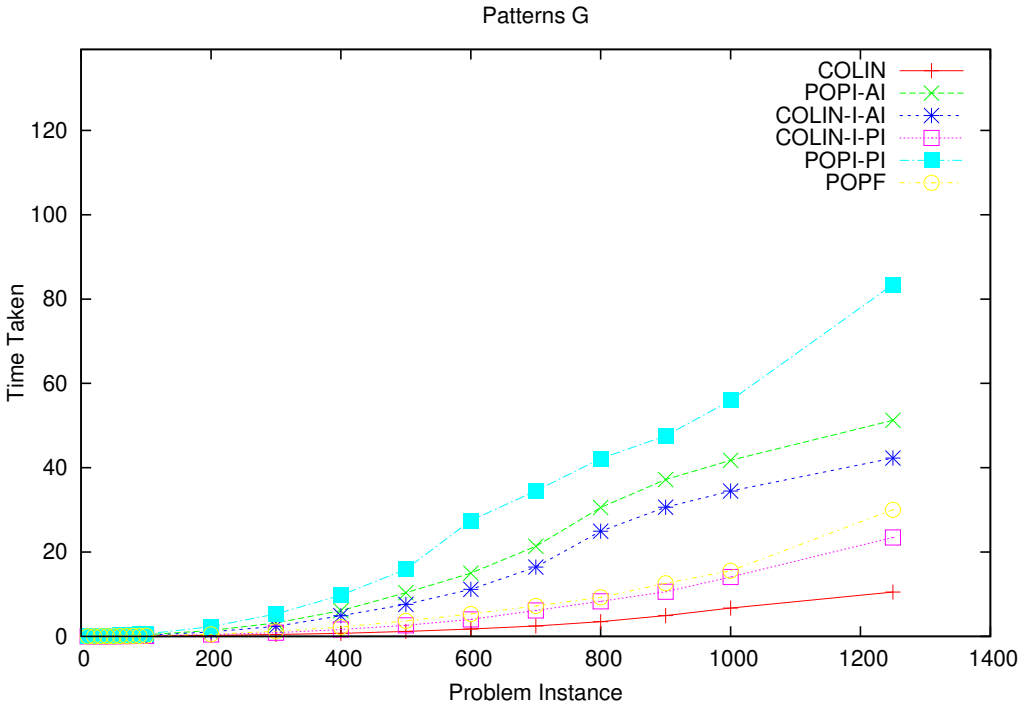



(b) Time Taken

Figure 7.27: Pattern A.

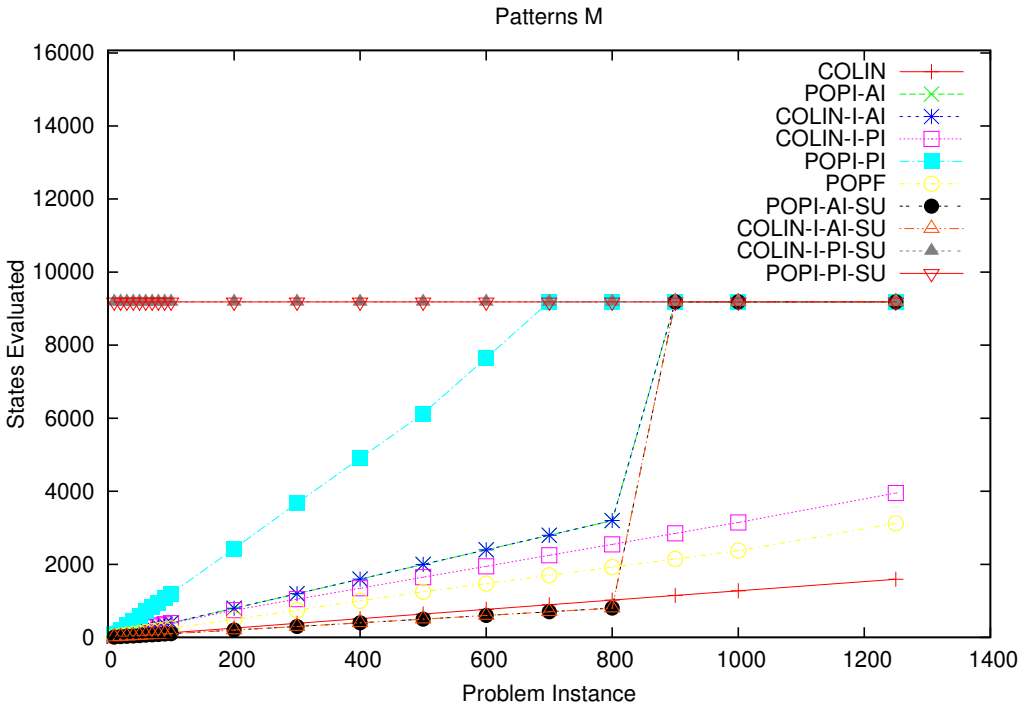



(a) States Evaluated

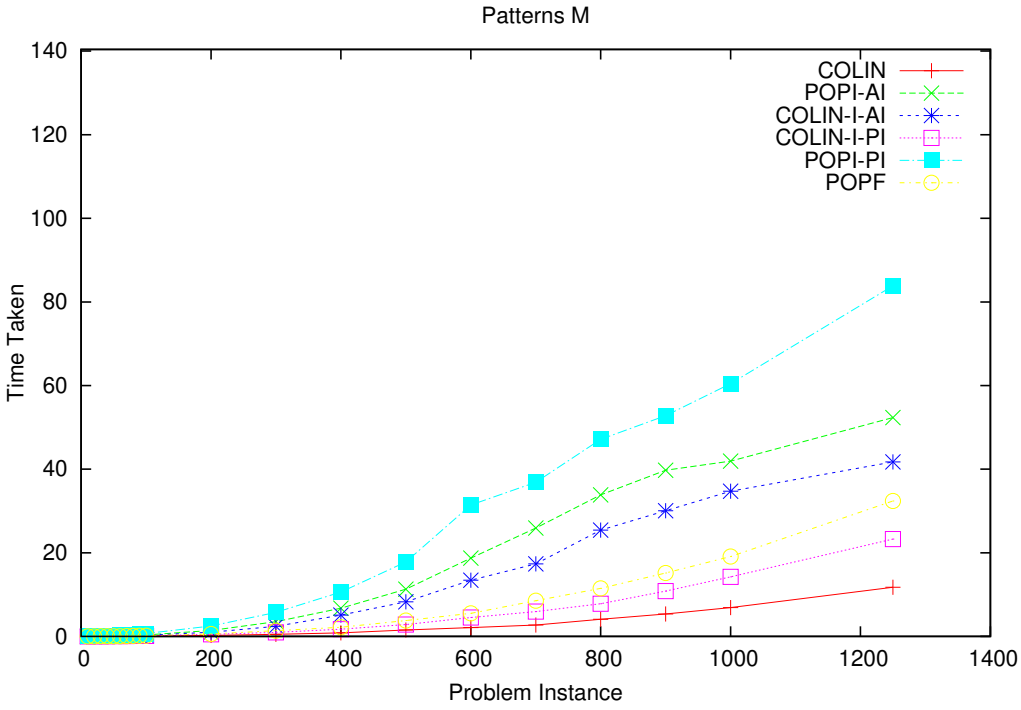



(b) Time Taken

Figure 7.28: Pattern G.

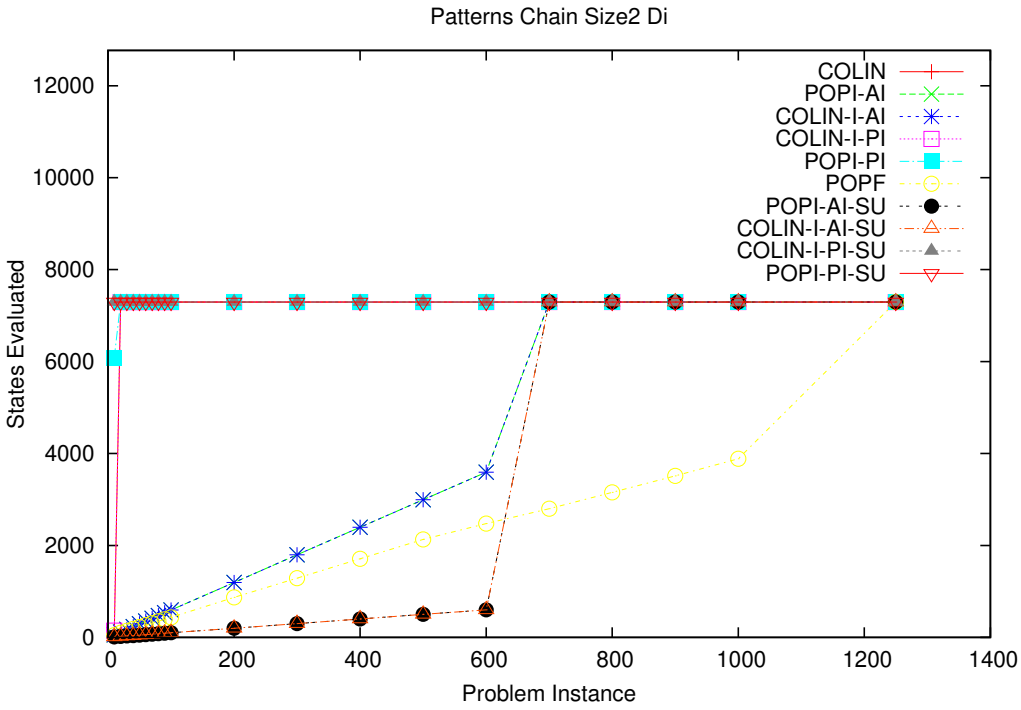



(a) States Evaluated

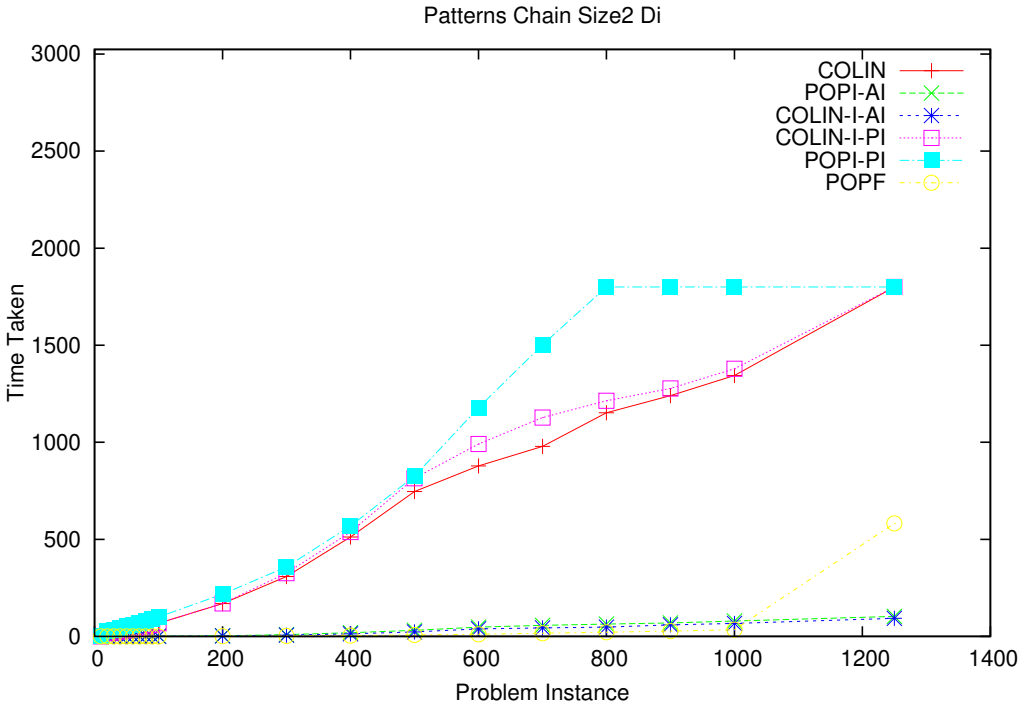



(b) Time Taken

Figure 7.29: Pattern M.

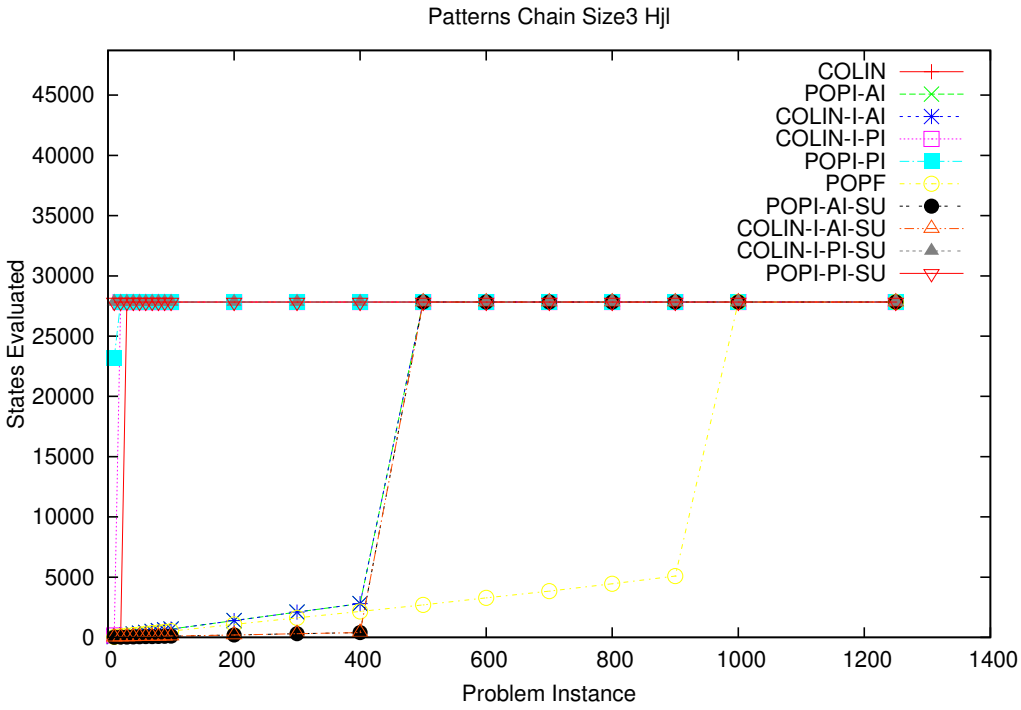



(a) States Evaluated

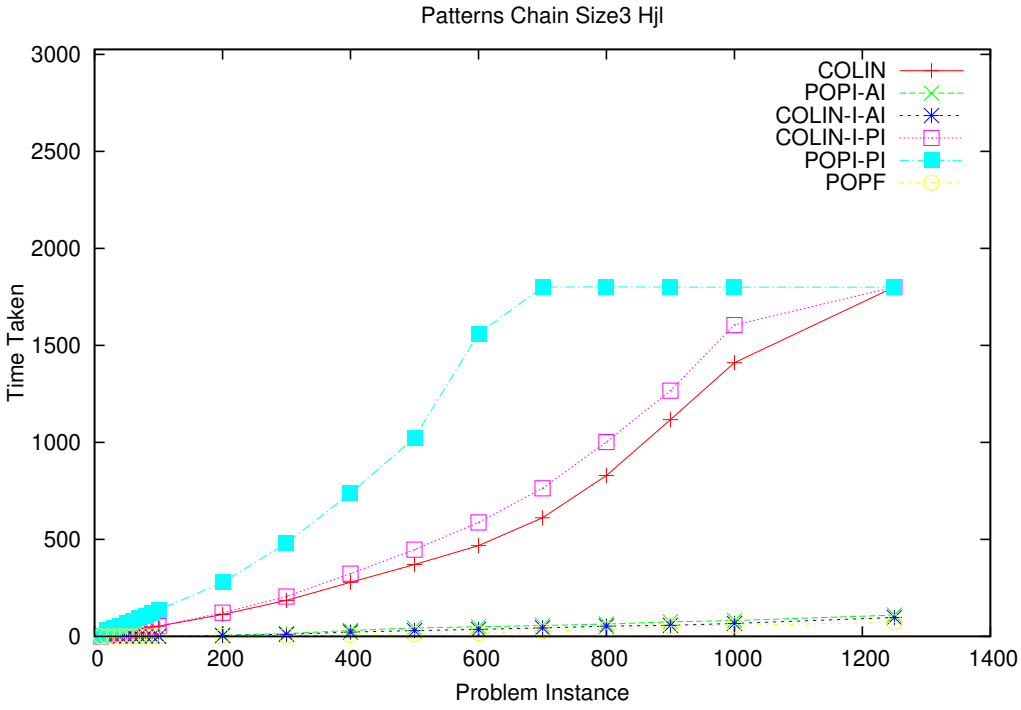



(b) Time Taken

Figure 7.30: Pattern Chain of size 2 containing types D and I.




(a) States Evaluated




(b) Time Taken

Figure 7.31: Pattern Chain of size 3 containing types H, J and L.

## 7.5 IPC Temporal Domains

In this section we perform experiments using the temporal domains submitted to the temporal satisficing track of the latest IPC in 2018. We first run the experiments using the POPF and POPI planners and compare the performance to measure any overhead costs incurred, from the machinery that POPI uses in addition to POPF's existing mechanisms. We then run temporal planners submitted to the same track as the domains in IPC 2018, to compare performance with POPF.

### 7.5.1 Comparison of POPI and POPF

In this section we provide results from running the aggressive and passive versions of POPI, and POPF on the temporal competition domains from IPC 2018. There were nine domains from which eight had no required concurrency between pairs of actions in them, hence no patterns can be detected in those domains. The purpose of running POPI and POPF on these domains is to provide data that shows that without the existence of pattern structures in the domain, the behaviour of POPI and POPF is the same.

#### Results

The results show that the specific problem instances solved amongst all the domains are the same for all three planners and that the number of states evaluated for the solved problems are also the same across all the planners. As these domains do not contain the pattern structures that drive POPI to do inference, this was the expected outcome and is supported by this result. In addition to confirming this hypothesis, another reason for performing these experiments is to determine what the other overhead costs are. The fact that the same problems are still solved even with POPI's additional memory usage, shows that POPI's general ability to solve other temporal problems without patterns of required concurrency is not diminished, given its extra usage of memory. The time taken to reach a solution or indeed run out of time or memory to solve a problem, is another indication of the overhead incurred by the POPI planners. We are interested in seeing what the overhead cost is for POPI as a result of performing the domain analysis, which POPI always performs to determine whether the pattern structures it deals with exist or not. For this reason we plot the results for the time taken to output solutions for the solved problems and the time taken for each of the planners to time out on the unsolved problems. This will give us an indication of the performance cost paid by the two versions of POPI for its pattern detection operations.

#### Analysis

Figures 7.32 and 7.33 show the performance time results of POPF versus the aggressive and passive versions of POPI respectively. We can observe that for both the aggressive and

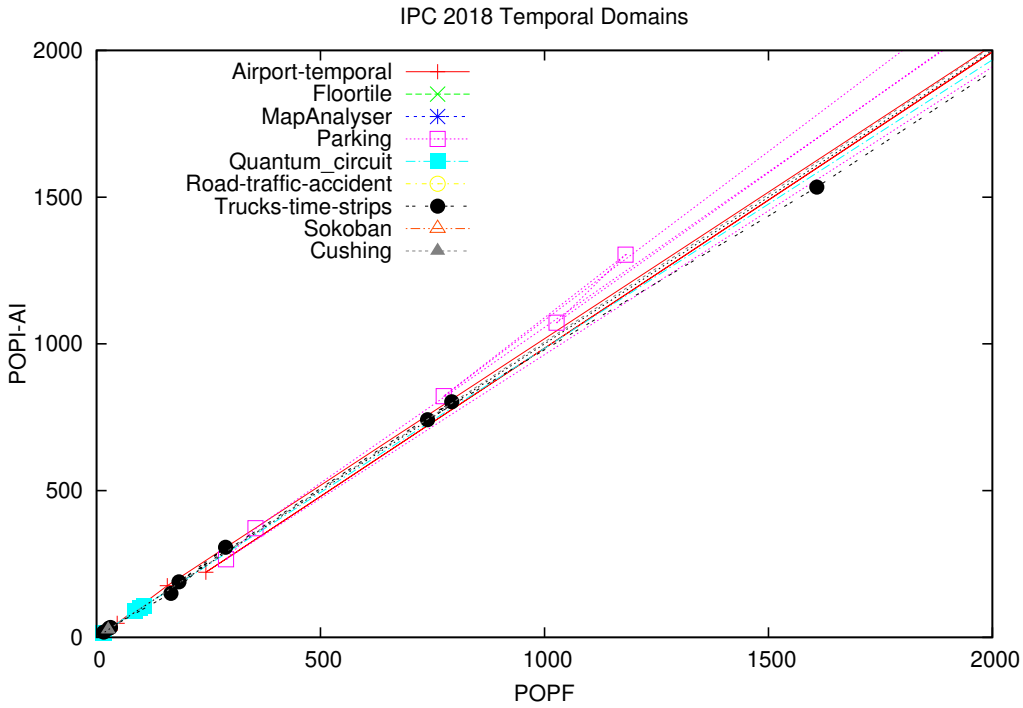



Figure 7.32: POPF compared with POPI-AI - Time Taken.

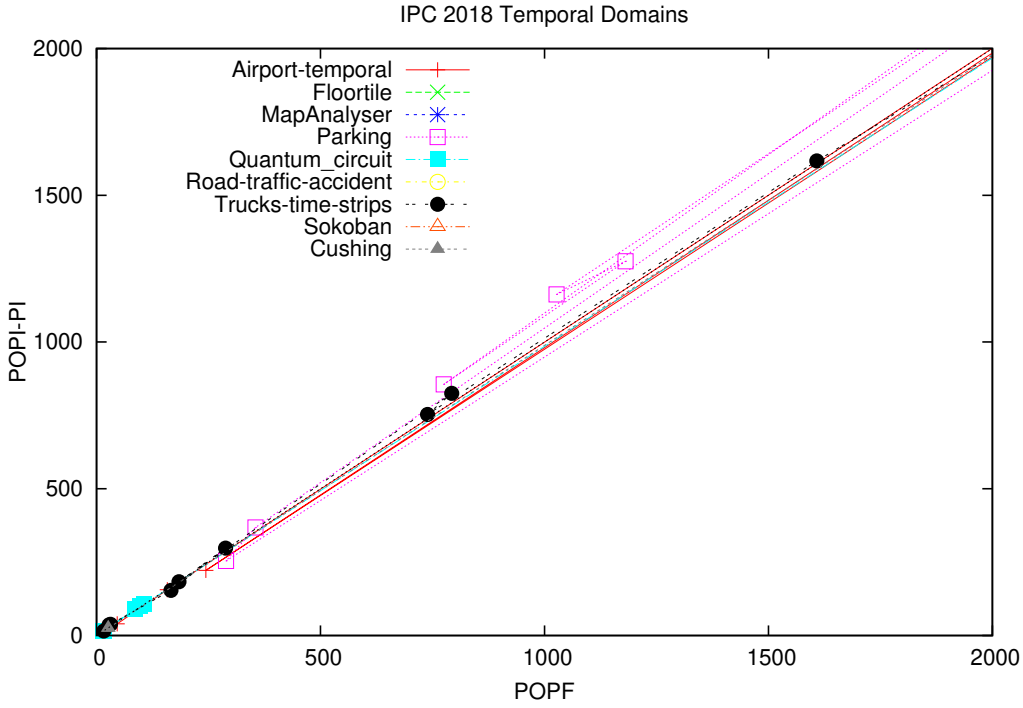



Figure 7.33: POPF compared with POPI-PI - Time Taken.

passive versions of POPI, the overhead in solving the problems in terms of solution time is very similar to that of POPF. There is some slight variation, and the time taken for the planners to time out on the problems not solved have showed similar results, indicating that for the problems where this was under the 30 minutes time limit, that all of the planners ran out of memory to solve the problems after similar amounts of time. The results for the Parking domain problems are somewhat anomalous for both versions of POPI, indicating that POPI may have analysed action structures in this domain where there were more candidates for required concurrency which did not turn out to be actual patterns instances. However, the solution time difference compared to POPF is less than 11%, which is within the margin of experimental error for these experiments. The results presented in this section show that POPI solved the same 29 problems across the IPC domains with comparable peformance, this satisfies the third evaluation objective presented at the start of this chapter.

### 7.5.2 Comparison of POPF and Other Temporal Planners

In this section we perform experiments using POPF and other temporal planners, most of which took part in the IPC 2018 temporal satisficing track. We again perform experiments using the temporal domains from the competition. The purpose of this comparison to to determine the competitiveness of POPF and in turn with POPI, with other recent temporal planners. We should note that each of these planners operate differently and produce different output. Since they are not all state-based planners, we cannot compare using number of states generated or evaluated. The time measurements outputted by each planner also varies in what is being measured. For example TPSHE, TP(K) and STP(K) output "Actual Search Time" values, whereas POPF outputs a "Time Taken" measurement. In addition, some of the planners also output multiple solutions. Since our comparison of the IPC temporal planners are with POPF, which produces a single solution, we use the time value produced for the first solution produced by each planner. Given these variations, the comparison of the planners based on the time to solve problems is not entirely fair. However, the results are still useful to map to give us an idea of the general performance of the planners across a range of temporal domains. A select number of the problem instances from each submitted domain to the IPC were used in the competition, which are the problems we have run our experiments on. The solution times produced in results for TFD are almost all the same, suggesting they may be time intervals within which a solution was found. This makes the TFD results difficult to compare with the other planners, therefore we present them separately in Section 7.5.3.

#### Results

The results mapping the performance of the planners on each of the IPC temporal domains are presented in Figures 7.34 to 7.42. A maximum value of 1800 seconds (30 minutes) is used on the graphs to represent problems which could not be solved, since there was no planner that solved a problem on the 30 minute deadline. The number of problems solved by each planner in each domain is summarised in Table 7.2, with the planners labelled as P1 to P11, and the corresponding planner names in Table 7.3. We can see from Table 7.2 that the number of problems solved by each planner varies between domains with TP-3 solving the largest total number of problems across the domains and STP-4 solving the least number of problems. No planner was able to solve problems from all nine domains, however most planners were able to solve all 10 problems in at least one domain. The `cushing` domain is the only domain that appears to have required concurrency. We observe that problems from this domain are solved by the TFLAP planner which was not able to solve problems in our pattern domain tests in Section 7.3.4. However, the cushing domain is defined with fixed duration actions, whereas the patterns domains use duration inequalities, indicating that this may have been the issue. The `cushing` domain contains actions corrresponding to pattern type B required concurrency as defined in this thesis. Therefore, it could also be that TFLAP can solve problems with

particular pattern structure but not others. For POPF, we can see that it solves 29 problems in total across the domains. Although some of the planners solve more problems than POPF, we can see that it is still competitive in general, as POPF is able to solve more problems than the STP planners. Interestingly, the `trucks` domain is one where each planner either solved all ten problem instances or none of them. The time measurements in Figure 7.42 show that the planners which did solve the problems, did so quickly, with the exception of TFLAP for one problem instance, where it took relatively much longer compared to its performance on the other problems for this domain.

### Summary

The purpose of the experiments comparing POPF to other recent temporal planners, was to show that POPF is still a relative and effective temporal planner. The results have shown that there are planners solving temporal problems that POPF has not, but POPF is still able to solve at least problems across most of the domains. The results illustrate the point from the fourth evaluation objective listed at the beginning of the chapter, that the baseline planner POPF used in this thesis, is able to compete with other temporal planners in solving temporal problems.

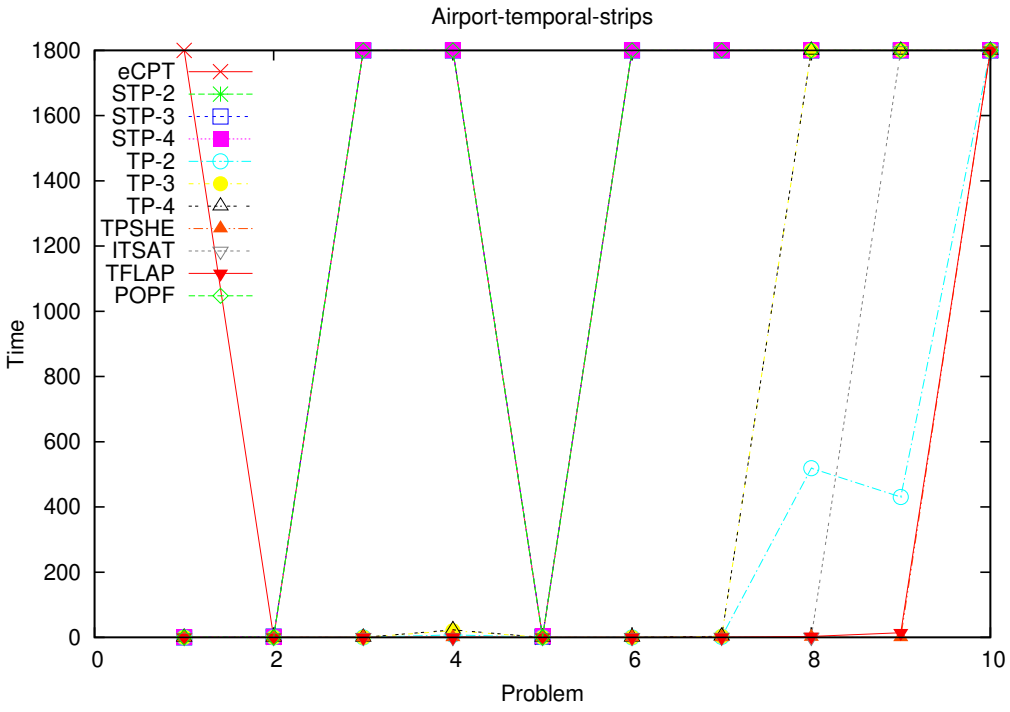



Figure 7.34: Airports Domain

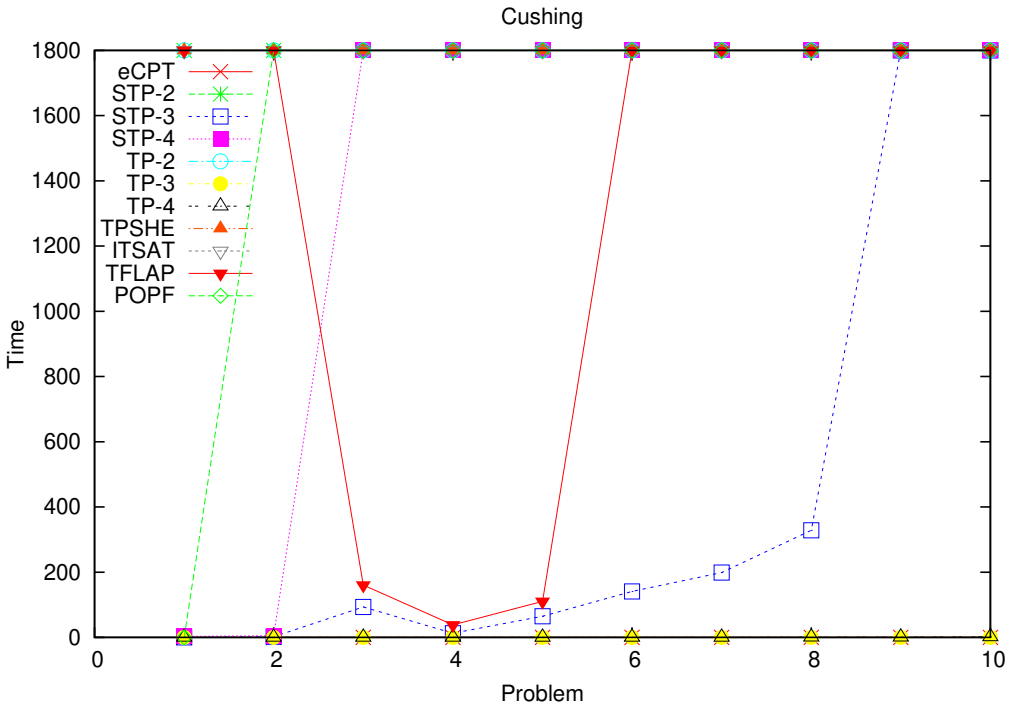



Figure 7.35: Cushing Domain

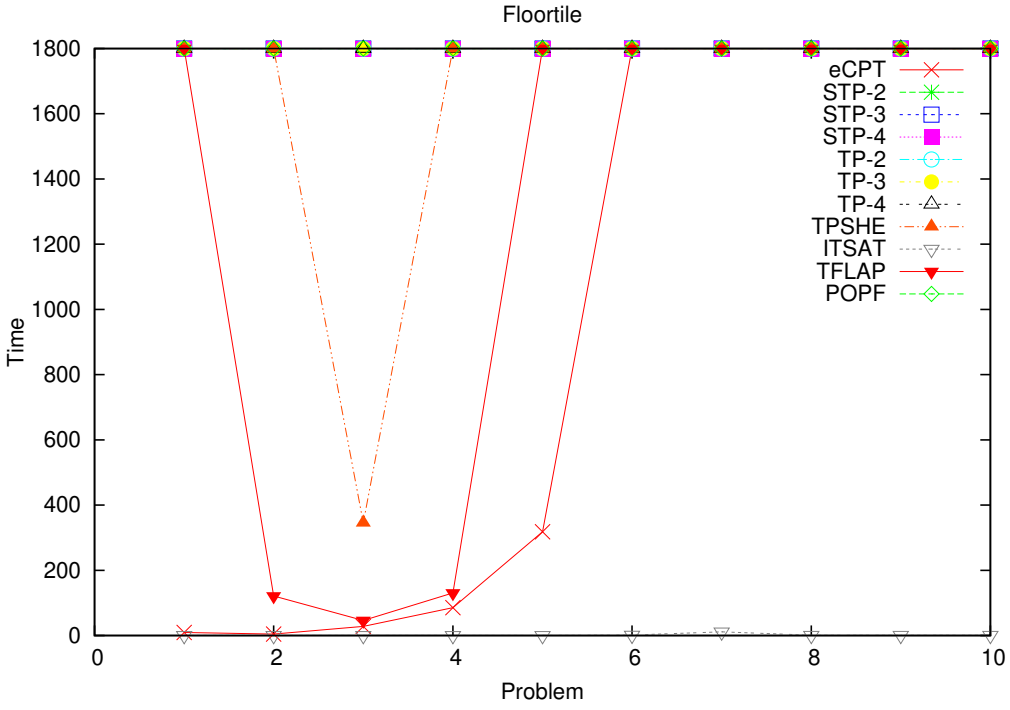



Figure 7.36: Floortile Domain

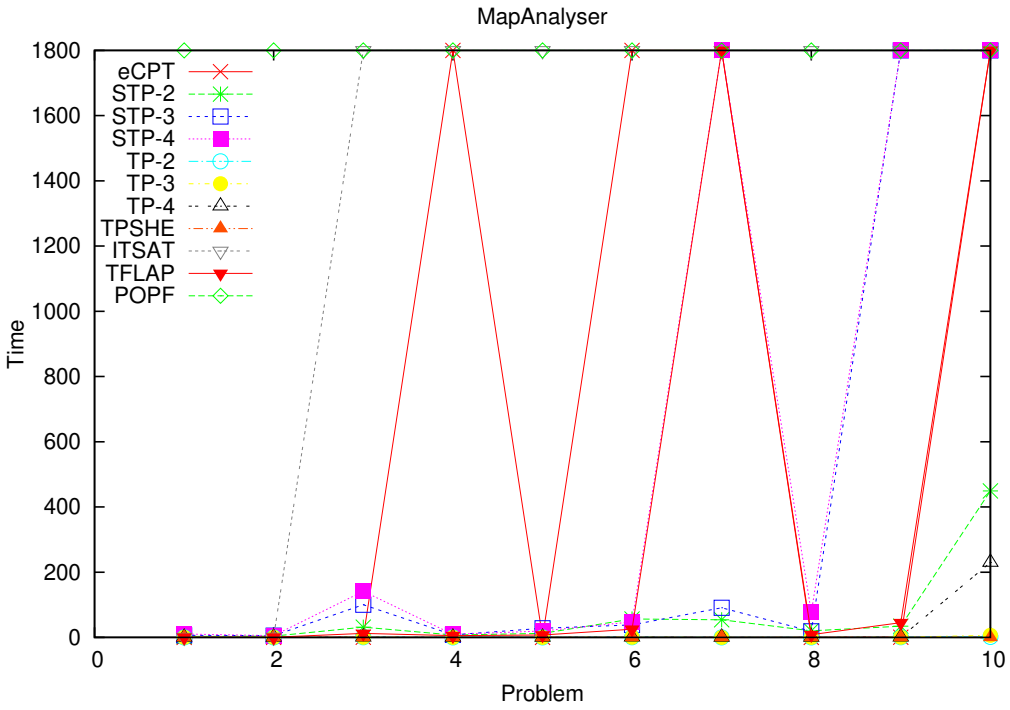



Figure 7.37: MapAnalyser Domain

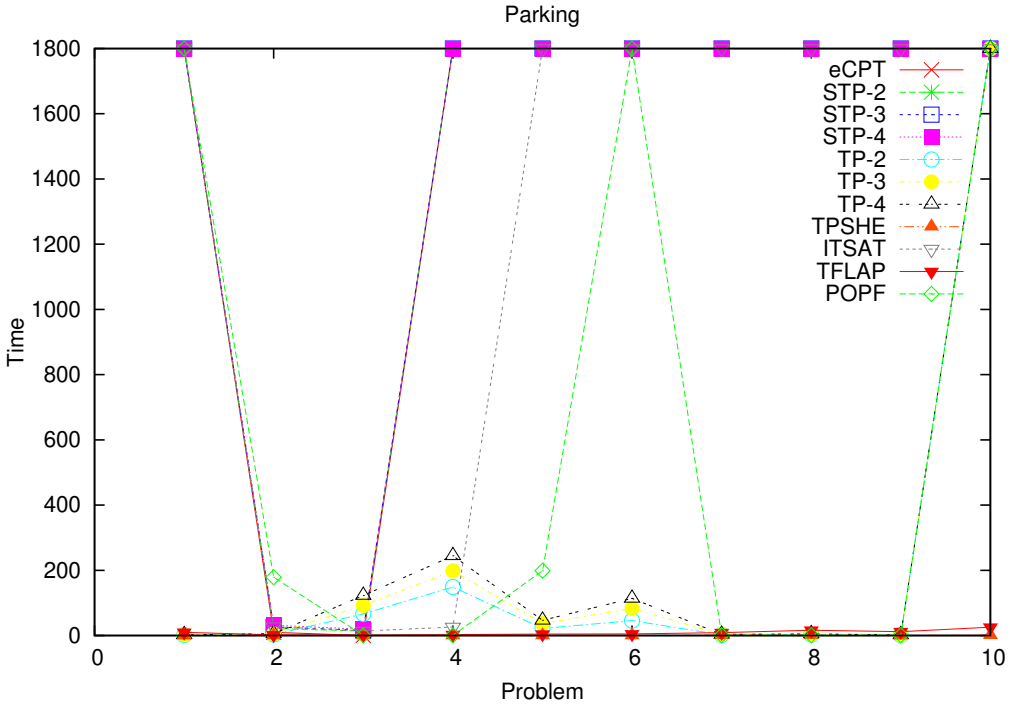



Figure 7.38: Parking Domain

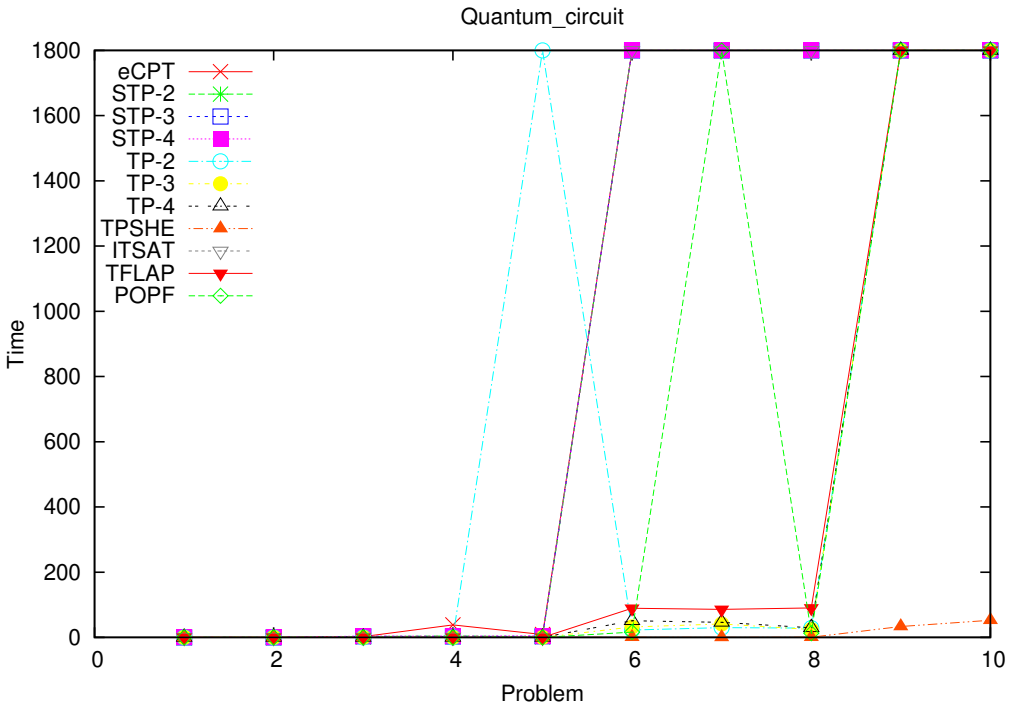



Figure 7.39: Quantum Circuit Domain

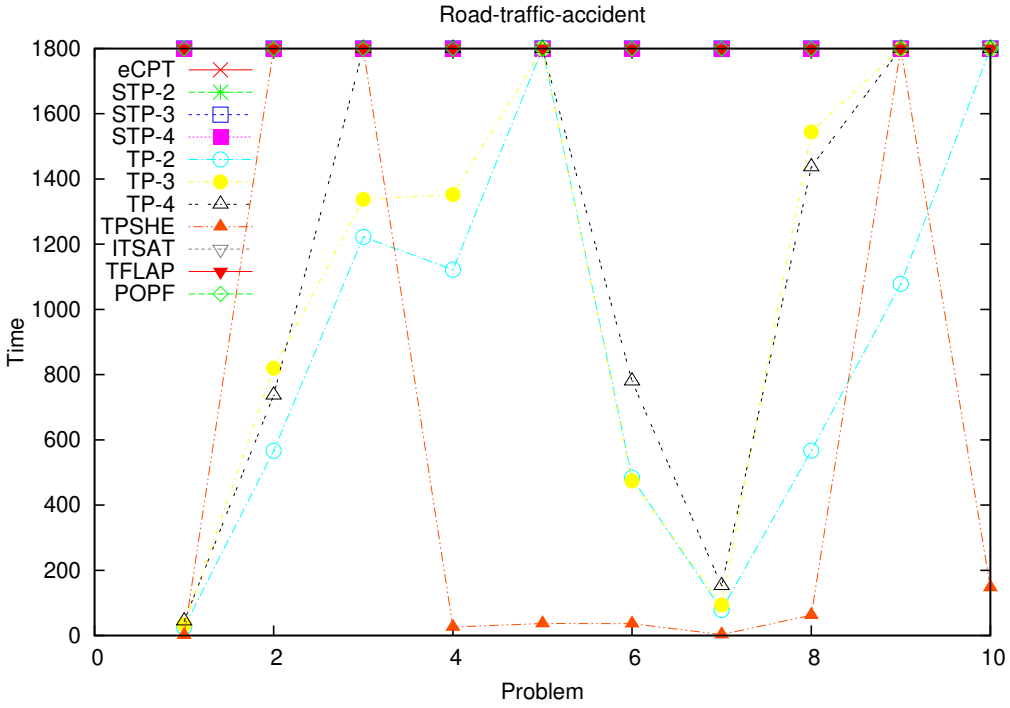



Figure 7.40: Road Traffic Accident Domain

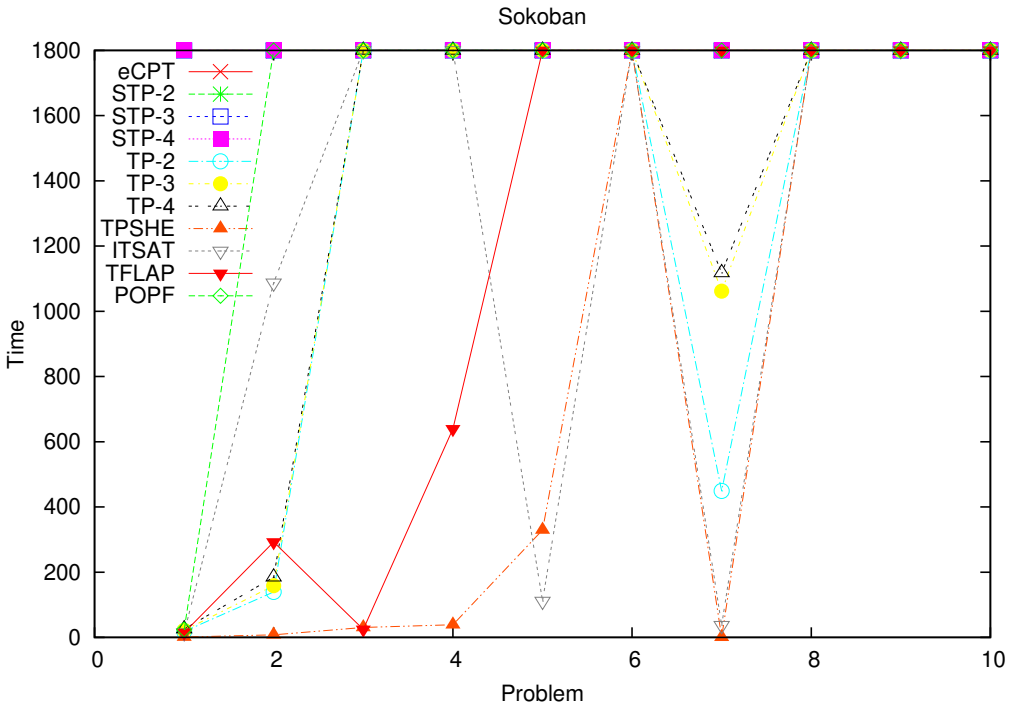



Figure 7.41: Sokoban Domain

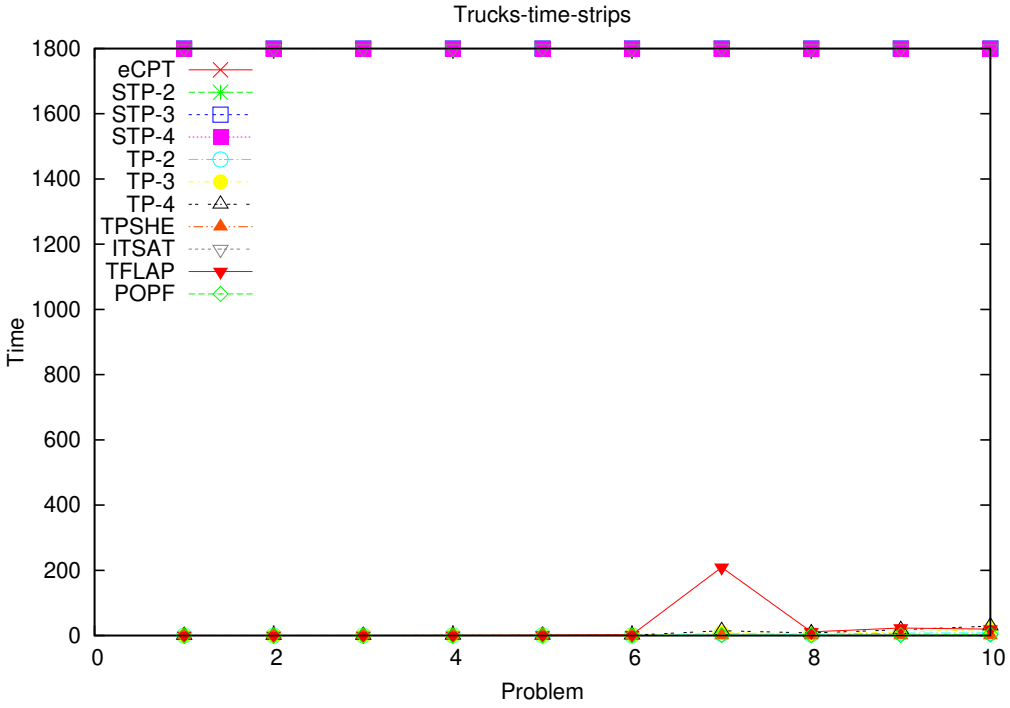



Figure 7.42: Trucks-time-strips Domain

| Planner | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | P10 | P11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Airport | 2 | 3 | 3 | 3 | 9 | 7 | 7 | 9 | 8 | 9 | 3 |
| Cushing | 10 | 0 | 8 | 2 | 0 | 10 | 10 | 0 | 0 | 3 | 1 |
| Floortile | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 10 | 3 | 0 |
| MapAnalyser | 6 | 10 | 8 | 7 | 10 | 10 | 10 | 10 | 2 | 8 | 0 |
| Parking | 2 | 2 | 2 | 2 | 9 | 9 | 9 | 10 | 3 | 10 | 7 |
| Quantum Circuit | 5 | 5 | 5 | 5 | 7 | 8 | 8 | 10 | 5 | 8 | 7 |
| Road Traffic Accident | 0 | 0 | 0 | 0 | 8 | 7 | 5 | 7 | 0 | 0 | 0 |
| Sokoban | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 6 | 4 | 4 | 1 |
| Trucks-time-strips | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 0 | 10 | 10 |
| Total Solved | 30 | 20 | 26 | 19 | 56 | 64 | 62 | 63 | 32 | 55 | 29 |

Table 7.2: Number of problems solved in each IPC domain by each planner.

| **P1** | **P2** | **P3** | **P4** | **P5** | **P6** |
|---|---|---|---|---|---|
| eCPT | STP-2 | STP-3 | STP-4 | TP-2 | TP-3 |
| **P7** | **P8** | **P9** | **P10** | **P11** | |
| TP-4 | TPSHE | ITSAT | TFLAP | POPF | |

Table 7.3: Label Key for Planners in Table 7.2

### 7.5.3 TFD on IPC Temporal Domains

In this section we present and analyse the results produced by TFD on the IPC temporal domain problems. TFD is a planner that produced mutiple solutions during experiments, however the search time extracted for the first solution was the same for most of the problems it solved. The results indicate that the initial search time outputted may be a bound on a time interval within which a plan is produced and not the "actual time" to produce the first solution to each problem.

#### Results

The results for each experiment run with TFD on the IPC temporal problems is presented in Table 7.4. The planner produced solution to a total of 49 of the 90 problems From the problems solved, 43 of them outputted 10 seconds as the search time, when the first plan was found. The other six problems appear to produce varying times for search in order to find solutions which seem more "realistic". Inspection of the results showed that plans were indeed produced for all 49 problems, however we have seen in the results produced by TFD in Section 7.3.3 that TFD has produced an invalid plan for a patterns domain problem containing required concurrency. For this reason it is difficult to the compare performance of TFD against the other temporal planners tested on the IPC temporal domains, as we cannot reliably assess the meanning of its results. However, if the 49 solutions are correct, then TFD is indeed a competitive planner at least in terms of the total number of problems solved.

| Problem Instance | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Airports-temporal-strips | 0.01 | 0.01 | 10 | 10 | 0.01 | 10 | 10 | ✗ | 10 | 10 |
| Cushing | 2.81 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Floortile | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| MapAnalyser | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Parking | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Quantum Circuit | 0.28 | 3.15 | 10 | 10 | 10 | 10 | 10 | 10 | ✗ | ✗ |
| Road Traffic Accident | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Sokoban | 10 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Trucks-time-strips | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |

Table 7.4: Search times (seconds) for initial solution produced by TFD.

#### Summary

The results produced by TFD which have been presented in this section do not allow us to draw a concrete conclusion as to its performance. This is since we were not able to find the type of time measurement needed from the results. This makes it difficult to compare

with other temporal planners tested. However, the total number of problems solved indicate that TFD can generally compete with other temporal planners and still solve many temporal problems.

## 7.6 Temporal Tea Domain

In the patterns domains, we have seen how the behaviour and performance of POPI and POPF can differ on large scale problems. In this section we look to illustrate the subtle differences in behaviour between the aggressive and passive inference strategies of POPI and how it can vary from the behaviour of POPF. In order to do this, we use a simple problem instance from the `temporalTea` domain that illustrates these differences. The goal of the problem in this domain is to be at home with a cup of tea made. The possible ways of achieving this are to go through the steps to make the tea at home, or to go to the café to buy some tea and come back home with it. The obvious choice is to make the tea at home but we will see how the planners deal with these options. The domain and problem instance used to run the test on each planner is provided in Appendix E.

### 7.6.1 Results

The plans in Figure 7.43 are the three solutions produced by POPI-AI, POPI-PI and POPF for a simple problem where the goal is to be at home with a cup of tea made. We can see that POPF first tries to perform an action in order to make the tea at home and then changes its strategy to achieve the goal by going to the café and buying the tea. This causes POPF to have a longer plan makespan and an extra action in the plan that is not needed. POPI-AI and POPI-PI produce the same plans, however POPI-PI used 3 state evaluations, whereas POPI-AI used only 2 state evaluations. This means that the first state generated by POPI-PI using the trigger action for the pattern of actions `visitcafe` and `buytea` produced a successor with a better heuristic, so the planner pursued the inference. POPI-AI did not use this extra state evaluation and benefited more than POPI-PI as a result.

### 7.6.2 Discussion

Although the example plans produced from this temporalTea domain problem are simple, they illustrate a clear point. We can see that the aggressive and passive strategies each perform better under different circumstances. It is clear that the two strategies can be set up in specific problem scenarios that makes one strategy perform better than the other. This illustrates the No Free Lunch (NFL) (Wolpert and Macready [1997]) theorem implications for using inference in planning. If we cannot guarantee where the states are better and taking the planner closer to the goal, applying inference from a state using the aggressive approach will cost computation time and not reliably provide gain in performance. Therefore, we pay a cost on some problems in exchange for gains in performance on other problems. Using the aggressive strategy causes the planner to pay the cost of pursuing inference, before it has an indication on whether the state generated at the end of the inference, will be better than the current state. Conversely, the passive strategy is more cautious, and only pursues the use of

```
0.000:  (visitcafe greentea) [25.000]
0.001:  (buytea greentea) [15.000]
```

(a) POPI-AI Plan

```
0.000:  (visitcafe greentea) [25.000]
0.001:  (buytea greentea) [15.000]
```

(b) POPI-PI Plan

```
0.000:  (addteabag greenteabag favouritemug) [2.000]
2.000:  (visitcafe greentea) [25.000]
2.001:  (buytea greentea) [15.000]
```

(c) POPF Plan

Figure 7.43: Plans produced for temporalTea problem by each planner.

inference when the trigger action is estimated to provide a state closer to the goal, however the state at the end of the inference can still be worse. Hence, the passive strategy gains less in performance, but does so more reliably. The question remains in asking what is the distribution of "realistic" problems in which it is better to apply one strategy or the other, and this remains an open question.

## 7.7 Summary

In this chapter we have provided an empirical analysis of the practical benefit that the infer and search process of POPI can bring, as well the costs, depending on the circumstances. It is evident from the results of the patterns domains that there are huge potential benefits of the aggressive inference approach when there is a repeated need to apply actions in either a single or chained set of pattern instances and when the use of this pattern based inference progressively leads the planner closer to the goal. Under these circumstances, POPI-AI is able to scale very well and better than POPF or COLIN or indeed the passive version of POPI (POPI-PI). This enabled us to achieve our first evaluation objective. We saw that eCPT, which is a constraint programming based planner that utilises inference, is not able to handle large scale problems for the Patterns domains. The experiments using TFD on this set of domains showed that it is not a planner designed for solving problems of required concurrency for most of the pattern types we deal with in this thesis. In addition, we used other recent temporal planners, which competed in the IPC 2018, and found that they were

limited in being able to handle our pattern domain problems. None of the pattern problems defined with duration inequalities were solved. This study achieve our second evaluation objective. The Optional concurrency domains containing patterns, showed us that when there is optional concurrency POPF is more competitive in its strategy. These experiments further illustrate that it very much depends of the exact circumstances of the problems being tackled, and the strategies being used, that determines which planner has the best performance. The `temporalTea` domain provides further evidence that the benefit from inference is still subjective, and depends on the strategy being used to apply it. We have also provided supporting evidence that where there is no pattern structure detected in the domain for the problem being solved, POPI in both versions has the same behaviour as POPF. This was achieved by comparing both variants of POPI against POPF on the IPC temporal domains, which showed that the planners all solved the exact same problems. This analysis achieved our third evaluation objective. We also performed a comparison of POPF against recent temporal planners from the IPC 2018 on the IPC temporal domain problems, which showed that POPF is generally able to compete with these planners in solving temporal problems. This allowed us to satisfy our fourth and final evaluation objective. In Chapter 8 we conclude the thesis by reviewing the work performed, its achievements and laying out areas for future extensions of the research.

# Chapter 8

# Conclusions

## 8.1 Summary

In this thesis we have investigated the notion of required concurrency between pairs of actions, both where there are only concurrent solutions and when there is a sequential solution with a concurrent alternative. In Chapter 3, we have catalogued a set of pattern instances where the predicate structure has been constructed such that each instance corresponds to one of sets of pattern sequences presented in Table 4.5 in Chapter 4. In Chapter 4 we have presented theorems and proved that the sets of pattern sequences in Table 4.5 are complete and exhaustive for pair-wise required concurrency. We have also proposed a novel method for measuring inferential gain using long established principles of information theory. We have developed a new planning system, called POPI, to explore the benefits of recognising the patterns presented in Chapter 3 and using an inference engine to power the use of more inference during planning.

It is clear from the results in Chapter 7 that there is a subclass of problems in temporal planning in which the pattern detection and inference strategies used by POPI provide an empirical gain over its baseline POPF. Depending on whether it is the aggressive or passive strategy being utilised, POPI optimistically or cautiously pursues its use of the pattern based inferences. It is important to remember that although POPI prioritises the use of its inference over search where possible, it is of course still a planner that uses search as well as the pre-existing inference machinery implemented in POPF.

The extent to which our new inferences or the pre-existing search or indeed a mixture of the two, is used in solving a problem depends on the problem being solved and its domain. If the domain contains required concurrency where two actions are detected as being in a pattern, then POPI is able to apply those actions using inference, once its trigger component has been applied via search. When a trigger action is applicable in a state, POPI prioritises

the trigger action over all others and treats them as the most helpful actions of all. This is because applying two, or in the case of a chain, many actions using inference and only evaluating the resulting state at the end, allows POPI to look ahead further into the search space. This results in the evaluating the potential benefit at a deeper level of the search tree, rather than the helpfulness of a single action, one state expansion at a time.

## 8.2 Future Directions for Research

There are various directions in which the research presented in this thesis could be extended. A natural extension which would follow nicely would be to consider action durations in addition to the predicate structure to handle certain subtle cases, for patterns such as type H, where the application of both endpoints of one action must applied in between the endpoints of another action. If there was a pair of actions with fixed durations where one action is shorter than the other and the pair are in a pattern H structure, how could the planner detect that only one of the two concurrent orderings is temporally viable, before applying the actions. Another interesting avenue to investigate would be how the patterns of required concurrency we have explored, could be extended to sets of sequences with more than two actions.

Currently in both versions of POPI, aggressive and passive, the pattern detection machinery utilises a technique of "predicate counting" when analysing the domain structure for candidate pattern instances. This was described in Section 6.4.3 of Chapter 6. The limitation of this approach is that, although it does guarantee that required concurrency between pairs of operators does exist when a pattern is detected, it is possible that even with a predicate part of a pattern structure existing in the schema of another operator, the required concurrency relationship may not be broken between the first two actions. This predicate counting system takes a safe and certain approach to detecting required concurrency, but can be restrictive in certain cases where the pattern structure does exist but because a pattern predicate exists outside of the action pair, the pattern detection is discarded. This is an area of future work that we would like to explore further, to look into ways of making the detection more accommodating, while maintaining the current safety of detecting pattern structures.

In Chapter 4 we explored the idea of being able to compile pairs of durative actions that are in a pattern of required concurrency into a single action under certain circumstances. Where such a case exists, that pair of grounded actions would be considered pattern abstract safe, meaning the preconditions and effects of the two actions can be compiled into one action. This core focus of this thesis has been on the interaction of the actions within the pattern and the constraints that exist from their specific required concurrency relationships. It would be fascinating to explore a technique for a planner which performs this compilation for action

pairs that are pattern abstract safe and investigates the benefits of this approach, compared with applying those same patterns using the inference based approach described in this thesis.

For the passive strategy, given that the first successor generated from applying the trigger action in a pattern is evaluated, an observation to note is that in cases where that state is not simply worse than its parent but is also a dead-end, there may be ways to prevent the use of such pattern instances again. In the current approach of POPI for both the passive and aggressive inference strategies, commitment to the application of the pattern of actions can be abandoned if it is deemed to be heuristically worse or a dead-end. This pattern instance is recorded to prevent it being triggered again from the same state and stop the planner becoming stuck in repeated applications of the same action pair. However, the pattern instance can be applied from a different state which may or may not be useful. Another interesting avenue of further research would be to see if this technique could be extended to determine situations, in which the planner could detect when no grounded action pair making up a pattern instance is usable to achieve the goal, after it being applied once. For example, perhaps there is a grounded atomic propositional fact defined in the set of goal conditions that is always deleted by all groundings of an action which is part of a pattern instance, hence applying any grounding of that pattern would always lead to a dead-end state. During the course of this research project, we took a preliminary step in this direction by simply preventing any re-use of any pattern instance where one grounding of that pattern led to a dead-end state. However, this has the potential to be over restrictive, as doing this without more information may prevent a useful re-application of the pattern instance with different grounded actions, for example at a different state. It is for this reason that this functionality is not currently set as a standard feature of POPI.

Another area of extension is also modifying both the aggressive and passive versions of POPI so that they no longer evaluate intermediary pattern states, as there heuristic values are not used during the execution of EHC-AI and EHC-AI strategies. This could allow an investigation into the savings from avoiding heuristic evaluation at these states.

Finally, it would also be interesting to look into how the ideas and approaches for pattern detection and inference techniques explored in this thesis could be extended to the temporal numeric case. Where there are temporal-numeric actions that also have concurrent temporal constraints, it would be valuable in the pursuit of using more inference in planning, to explore if there is a new way to infer values of numeric variables. This could work by leverage what POPI already knows about required concurrency relationship between actions, due to precedence constraints created by the temporal structure of the action pair, to see if information about the numeric effects could be inferred in certain circumstances.

# Appendix A

# Mars Rover Domain

## A.1 Domain

This domain illustrates the temporal coordination problem in which actions must be coordinated to occur at certain times in order to solve the problem.

```
(define (domain marsRover)
(:requirements :strips :typing :fluents :equality :durative-actions)
(:types waypoint rover drone objective sensor camera fuelStation)
(:predicates
(at ?r - rover ?w - waypoint)
(lightAt ?w - waypoint)
(launched ?d - drone)
(inAir ?d - drone)
(droneAt ?d - drone ?w - waypoint)
(inspected ?w - waypoint)
)

(:durative-action navigate
 :parameters(?r - rover ?w1 ?w2 - waypoint)
 :duration(= ?duration 10)
 :condition (at start (at ?r ?w1))
 :effect(and (at start(not(at ?r ?w1)))
 (at end (at ?r ?w2))
)
)

(:durative-action launch
```

```
 :parameters(?d - drone)
 :duration(= ?duration 2)
 :condition (and)
 :effect(and (at start(inAir ?d))
  (at end (not(inAir ?d)))
)
)

(:durative-action fly
 :parameters(?d - drone ?w1 ?w2 - waypoint)
 :duration(= ?duration 1)
 :condition (and (at start (droneAt ?d ?w1))
 (at start(inAir ?d))
)
 :effect (and (at start(not(droneAt ?d ?w1)))
  (at end (droneAt ?d ?w2))
)
)

(:durative-action shine_light
 :parameters(?r - rover ?w - waypoint)
 :duration(= ?duration 1)
 :condition (over all (at ?r ?w))
 :effect (and (at start(lightAt ?w))
  (at end (not(lightAt ?w)))
)
)

(:durative-action inspect
 :parameters(?d - drone ?w - waypoint)
 :duration(= ?duration 0.5)
 :condition (and (over all (lightAt ?w))
     (over all (inAir ?d))
     (over all (droneAt ?d ?w)))
 :effect (and (at end (inspected ?w))
 )
)
)
```

## A.2 Problem Instance

```
(define (problem marsProb)
(:domain marsRover)
(:objects turtle - rover
          parrot - drone
  w1 w2 w3 - waypoint
)

(:init
(at turtle w1)
(droneAt parrot w2)
)
(:goal (inspected w3))
)
```

# Appendix B

# Patterns Type D Domain

## B.1 Domain

This domain is one version of the Patterns domain that contains a single instance of the pattern D structure between actions A and B. Other versions of the domain include the same with modifications to actions A and B to model each of the other pattern types. This domain along with the other *Patterns* domains are artificial domains which have be made to illustrate how POPI works and the benefits its approach can give.

```
(define (domain patternsD)
(:requirements :strips :typing :equality :durative-actions
 :duration-inequalities)
(:types typeA)
(:predicates
(p ?a - typeA)
(q ?b - typeA)
(r ?d - typeA)
(next ?o1 ?o2 - typeA)
(ready ?o1 - typeA)
(active)
)

(:durative-action Act_A
 :parameters(?a ?b - typeA )
 :duration(and (<= ?duration 5) (>= ?duration 1))
 :condition(and (at start (active)) (at end (q ?a))
  (at start(next ?a ?b)) (at start (ready ?a)) (at end(ready ?a)))
 :effect (and (at start(p ?a)) (at start(not(active))) (at start(ready ?b))
```

```
    (at end (active)) (at end(not(ready ?a)))
   )
 )

(:durative-action Act_B
 :parameters(?a  - typeA)
 :duration(and (<= ?duration 5) (>= ?duration 1))
 :condition(and (at start(p ?a)) (at start (ready ?a)))
 :effect (and (at end (q ?a)))
)

  (:durative-action Act_C
 :parameters(?a ?b - typeA)
 :duration(and(<= ?duration 0.9 ) (>= ?duration 0.5))
 :condition(and (at start(ready ?a)) (at start(next ?a ?b))
  (over all (ready ?b))
  )
 :effect (and (at start(not(ready ?a))) (at start(ready ?b))
      (at end(not(ready ?b)))
   )
)

)
```

## B.2 Problem Instance

```
(define (problem patternsProb10)
(:domain patternsD)
(:objects  obj1 obj2 obj3 obj4 obj5 obj6 obj7 obj8 obj9 obj10 - typeA)

(:init
(ready obj1)
(next obj1 obj2)
(next obj2 obj3)
(next obj3 obj4)
(next obj4 obj5)
(next obj5 obj6)
(next obj6 obj7)
```

```
(next obj7 obj8)
(next obj8 obj9)
(next obj9 obj10)
(active)
)

(:goal (and (ready obj10)))

)
```

# Appendix C

# Patterns Type B Domain

## C.1 Domain (Fixed Durations)

This domain is a version of the Patterns domain containing a pattern type B structure. This domain has been specified with fixed durations for all the actions to accommodate testing on the eCPT planner. Section C.2 shows the problem instance used for testing this version of the Patterns domain.

```
(define (domain patternsB)
(:requirements :strips :typing :equality :durative-actions)
(:types typeA )
(:predicates
(p ?a - typeA)
(r ?d - typeA)
(next ?o1 ?o2 - typeA)
(ready ?o1 - typeA)
(active )
)

(:durative-action Act_A
 :parameters(?a ?b - typeA )
 :duration(= ?duration 5)
 :condition(and (at start (active)) (at start(next ?a ?b))
      (at start (ready ?a))  (at end(ready ?a))
      (at end (r ?a)) )
 :effect (and (at start(p ?a)) (at end(not(p ?a)))
    (at start(not(active))) (at start(ready ?b))
    (at end (active)) (at end(not(ready ?a)))
```

```
   )
 )

(:durative-action Act_B
 :parameters(?a  - typeA)
 :duration(= ?duration 4.5)
 :condition(and (at start(p ?a)) (at start (ready ?a)))
 :effect (and (at end (r ?a) ))
)

  (:durative-action Act_C
 :parameters(?a ?b - typeA)
 :duration(= ?duration 1 )
 :condition(and (at start(ready ?a)) (at start(next ?a ?b))
  (over all (ready ?b)))
 :effect (and (at start(not(ready ?a))) (at start(ready ?b))
    (at end(not(ready ?b))))
)

)
```

## C.2 Problem Instance

```
(define (problem patternsProb2)
(:domain patternsB)
(:objects obj1 obj2 - typeA)

(:init
(active)
(ready obj1)
(next obj1 obj2)
)

(:goal (and (ready obj2)))

)
```

# Appendix D

# MyBuilding Domain

## D.1 Domain (Version 1)

This domain and problem along with modified variations of it have been used to generate the example search space diagrams presented in section 6.8 of chapter 6. Variations of this domain model were made in order to illustrate the behaviour of each strategy in the different scenarios that were presented. The MyBuilding domain describes a scenario where there is a building, you are in one room location and are attempting to go to another location. There are two ways of travelling between the room you start in and the destination room. The first is to move between rooms that are connected by paths. The second is to make a hole in the wall to get to the destination room. The problem is that although that may seem like the better plan, making a hole in the wall makes the building unstable. The goal is to be both at the destination room and for the building to be stable.

```
(define (domain mybuilding)
(:requirements :strips :typing :equality :durative-actions)
(:types location)
(:predicates
(buildingStable)
(wall ?a - location ?b - location)
(at ?a - location)
(open ?a ?b - location)
(closed ?a ?b - location)
(path ?a ?b - location)
)

     (:durative-action make-hole
```

```
 :parameters (?x ?y - location)
 :duration (= ?duration 100)
 :condition (and (at start (at ?x)) (at start (wall ?x ?y))
     (at end (at ?y)))
 :effect (and (at start (path ?x ?y)) (at start (not (wall ?x ?y)))
  (at end (not (path ?x ?y))) (at end (not (buildingStable))))
)

(:durative-action go-through
 :parameters (?x ?y - location)
 :duration (= ?duration 50)
 :condition (and (at start (at ?x)) (at start (path ?x ?y))
   (over all (path ?x ?y)))
 :effect (and  (at start (not (at ?x))) (at end (at ?y)))
)

(:durative-action go-direct
 :parameters (?x ?y - location)
 :duration (= ?duration 10)
 :condition (and (at start (at ?x)) (at start (open ?x ?y)))
 :effect (and (at start (not (at ?x))) (at end (at ?y)))
)

(:durative-action open
 :parameters (?x ?y - location)
 :duration (= ?duration 1)
 :condition (at start (closed ?x ?y))
 :effect (and (at start (not (closed ?x ?y))) (at end (open ?x ?y)))
)

)
```

## D.2 Problem Instance

```
(define (problem mybuildingProb1)
(:domain mybuilding)
(:objects  sloc room1 dest - location)
```

```
(:init
(at sloc)
(buildingStable)
(open sloc room1)
(closed room1 dest)
(wall room1 dest)
)
(:goal (and (at dest) (buildingStable)))

)
```

# Appendix E

# Temporal Tea Domain

## E.1 Domain

This domain has been made to illustrate the difference in behaviour between the aggressive and passive versions of POPI and against POPF, depending on the problem being solved and the strategy being used. We also provide an example problem instance used in the experiments for this domain.

```
(define (domain temporalTea)
(:requirements :strips :typing :equality :durative-actions)
(:types mug tea teaBag milk water)
(:predicates
(addedTo ?m - milk ?mu - mug)
(atBottomOf ?t - teaBag ?m - mug)
(containedIn ?w - water ?m - mug)
(drinkMade ?t - tea)
(haveDrink ?t - tea)
(handempty)
(athome)
(atcafe)
(handdirty)
(atfrontdoorhome)
(nothanddirty)
)

(:durative-action visitcafe
:parameters (?t - tea)
:duration (= ?duration 25)
```

```
:condition (and  (at start (athome)) (at end (haveDrink ?t)))
:effect (and (at start (not (athome))) (at start (atcafe))
 (at end(athome)) (at end(drinkMade ?t)))
)

(:durative-action buytea
:parameters (?t - tea)
:duration (= ?duration 15)
:condition (and (at start (atcafe)))
:effect (and (at end (haveDrink ?t)))
)

(:durative-action getMilk
 :parameters(?m - milk ?mu - mug)
 :duration (= ?duration 2)
 :condition (and (at start (handempty)) (over all(athome)))
 :effect(and (at start (addedTo ?m ?mu)) (at start (not (handempty)))
      (at end (handdirty)) (at end (not (nothanddirty))))
)

(:durative-action addWater
:parameters (?w - water ?m - mug)
:duration (= ?duration 2)
:condition (and (at start (handempty)) (over all(athome)))
:effect(and (at start (containedIn ?w ?m)) (at start (not (handempty)))
(at end (handdirty)) )
)

(:durative-action addTeaBag
:parameters (?t - teaBag ?m - mug)
:duration (= ?duration 2)
:condition (and (at start (handempty)) (over all(athome)))
:effect(and (at start(atBottomOf ?t ?m)) (at start (not (handempty)))
(at end (handdirty)) (at end (not (nothanddirty))))
)

(:durative-action clean
:parameters ()
```

```
:duration (= ?duration 1)
:condition (and (at start (handdirty)) (over all(athome)))
:effect (and (at start (nothanddirty)) (at start (not (handdirty)))
 (at end (handempty)))
)

(:durative-action mix
:parameters (?t - teaBag ?w - water ?m - milk ?mu - mug ?te -tea)
:duration (= ?duration 3)
:condition (and (at start (handempty))(at start(addedTo ?m ?mu))
(at start(atBottomOf ?t ?mu)) (at start(containedIn ?w ?mu))
(over all(athome)))
:effect (and (at end(drinkMade ?te)) (at start (not (handempty)))
 (at end (handempty)))
)

)
```

## E.2 Problem Instance

```
(define (problem tempTeaProb)
(:domain temporalTea)
(:objects
  greenTeaBag - teaBag
  greenTea - tea
  someWater1 - water
  favouriteMug - mug
  soyaMilk - milk
)

(:init
(handempty)
(athome)
)

(:goal (and (athome) (drinkMade greenTea)))

)
```

# Bibliography

James F. Allen. Planning as temporal reasoning. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991.*, pages 3–14, 1991.

J. Benton, Amanda Jane Coles, and Andrew Coles. Temporal planning with preferences and time-dependent continuous costs. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, 2012. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4699.

Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 1636–1642, 1995. URL http://ijcai.org/Proceedings/95-2/Papers/080.pdf.

Blai Bonet and Hector Geffner. Planning as heuristic search: New results. In *Recent Advances in AI Planning, 5th European Conference on Planning, ECP'99, Durham, UK, September 8-10, 1999, Proceedings*, pages 360–372, 1999. doi: 10.1007/10720246_28. URL https://doi.org/10.1007/10720246_28.

Isabel Cenamor, Mauro Vallati, Lukas Chrpa, Tomas de la Rosa, and Fernando Fernandez. Temporal: Temporal portfolio algorithm. international planning competition. https://icenamor.github.io/files/TemPoRal.pdf, 2018.

Yixin Chen, Benjamin W. Wah, and Chih-Wei Hsu. Temporal planning using subgoal partitioning and resolution in sgplan. *J. Artif. Intell. Res.*, 26:323–369, 2006. doi: 10.1613/jair.1918. URL https://doi.org/10.1613/jair.1918.

A. J. Coles, A. I. Coles, M. Fox, and D. Long. Forward-chaining partial-order planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10)*, May 2010. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS10/paper/view/1421/1527.

Amanda Coles and Andrew Coles. A temporal relaxed planning graph heuristic for planning with envelopes. In *Proceedings of the Twenty Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017), June 18-23, 2017, Pittsburgh, USA*, 2017.

Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Temporal planning in domains with linear processes. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1671–1676, 2009a. URL `http://ijcai.org/Proceedings/09/Papers/279.pdf`.

Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Extending the use of inference in temporal planning as forwards search. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 2009b. URL `http://aaai.org/ocs/index.php/ICAPS/ICAPS09/paper/view/743`.

Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. COLIN: planning with continuous linear numeric change. *J. Artif. Intell. Res. (JAIR)*, 44:1–96, 2012. doi: 10.1613/jair.3608. URL `http://dx.doi.org/10.1613/jair.3608`.

Andrew Coles and Amanda Smith. Marvin: A heuristic search planner with online macro-action learning. *J. Artif. Intell. Res.*, 28:119–156, 2007. doi: 10.1613/jair.2077. URL `https://doi.org/10.1613/jair.2077`.

Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Planning with problems requiring temporal coordination. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 892–897, 2008. URL `http://www.aaai.org/Library/AAAI/2008/aaai08-142.php`.

Andrew Coles, Maria Fox, Keith Halsey, Derek Long, and Amanda Smith. Managing concurrency in temporal planning using planner-scheduler interaction. *Artif. Intell.*, 173(1):1–44, 2009c. doi: 10.1016/j.artint.2008.08.003. URL `http://dx.doi.org/10.1016/j.artint.2008.08.003`.

William Cushing, Subbarao Kambhampati, Mausam, and Daniel S. Weld. When is temporal planning really temporal? In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 1852–1859, 2007a. URL `http://dli.iiit.ac.in/ijcai/IJCAI-2007/PDF/IJCAI07-299.pdf`.

William Cushing, Daniel S. Weld, Subbarao Kambhampati, Mausam, and Kartik Talamadupula. Evaluating temporal planning domains. In *Proceedings of the Seventeenth*

*International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, pages 105–112, 2007b. URL `http://www.aaai.org/Library/ICAPS/2007/icaps07-014.php`.

Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artif. Intell.*, 49 (1-3):61–95, 1991. doi: 10.1016/0004-3702(91)90006-6. URL `https://doi.org/10.1016/0004-3702(91)90006-6`.

Giuseppe Della Penna, Daniele Magazzeni, Fabio Mercorio, and Benedetto Intrigila. Upmurphi: A tool for universal planning on PDDL+ problems. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 2009. URL `http://aaai.org/ocs/index.php/ICAPS/ICAPS09/paper/view/707`.

Minh B. Do and Subbarao Kambhampati. Sapa: a domain-independent heuristic metric temporal planner. *In Proc. 6th European Conference on Planning (ECP)*, pages 82–91, 2001.

Stefan Edelkamp and Jörg Hoffmann. Pddl2.2: The language for the classical part of ipc-4 - extended abstract. In *The 4th International Planning Competition Booklet*, pages 2–6, 2004. URL `http://ipc04.icaps-conference.org/deterministic/`.

Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 2009. URL `http://aaai.org/ocs/index.php/ICAPS/ICAPS09/paper/view/742`.

Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971. doi: 10.1016/0004-3702(71)90010-5. URL `https://doi.org/10.1016/0004-3702(71)90010-5`.

Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003. doi: 10.1613/jair.1129. URL `http://dx.doi.org/10.1613/jair.1129`.

Maria Fox and Derek Long. Modelling mixed discrete-continuous domains for planning. *J. Artif. Intell. Res.*, 27:235–297, 2006. doi: 10.1613/jair.2044. URL `https://doi.org/10.1613/jair.2044`.

Furelos-Blanco, A. D., Jonsson, H. Palacios, and S. Jiménez. Forward-search temporal planning with simultaneous events. In *Proceedings of the 13th Workshop on Constraint Satisfaction Techniques for Planning and Scheduling (COPLAS) at the International Conference on Automated Planning and Scheduling (ICAPS-18)*, pages 11–20, 2018.

D. Furelos-Blanco and A. Jonsson. Cp4tp: A classical planning for temporal planning portfolio. temporal track of the international planning competition (ipc). `https://ipc2018-temporal.bitbucket.io/planner-abstracts/team1.pdf`, 2018.

Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. An approach to temporal planning and scheduling in domains with predictable exogenous events. *J. Artif. Intell. Res.*, 25:187–231, 2006. doi: 10.1613/jair.1742. URL `https://doi.org/10.1613/jair.1742`.

Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.*, 173(5-6):619–668, 2009. doi: 10.1016/j.artint.2008.10.012. URL `https://doi.org/10.1016/j.artint.2008.10.012`.

K. Halsey, D. Long, and M. Fox. Crikey - a temporal planner looking at the integration of scheduling and planning. In *Proceedings of the Workshop on Integration Scheduling Into Planning at 13th International Conference on Automated Planning and Scheduling (ICAPS'03)*, pages 46–52, June 2004. URL `https://strathprints.strath.ac.uk/1948/`.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2): 100–107, 1968. doi: 10.1109/TSSC.1968.300136. URL `https://doi.org/10.1109/TSSC.1968.300136`.

Malte Helmert. A planning heuristic based on causal graph analysis. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, pages 161–170, 2004. URL `http://www.aaai.org/Library/ICAPS/2004/icaps04-021.php`.

Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246, 2006. doi: 10.1613/jair.1705. URL `https://doi.org/10.1613/jair.1705`.

Malte Helmert. Changes in pddl 3.1. 2008. URL `http://icaps-conference.org/ipc2008/deterministic/PddlExtension.html`.

Jörg Hoffmann. The metric-ff planning system: Translating "ignoring delete lists" to numeric state variables. *J. Artif. Intell. Res. (JAIR)*, 20:291–341, 2003. doi: 10.1613/jair.1144. URL `http://dx.doi.org/10.1613/jair.1144`.

Jörg Hoffmann and Stefan Edelkamp. The deterministic part of IPC-4: an overview. *J. Artif. Intell. Res.*, 24:519–579, 2005. doi: 10.1613/jair.1677. URL `https://doi.org/10.1613/jair.1677`.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.*, 14:253–302, 2001. doi: 10.1613/jair.855. URL https://doi.org/10.1613/jair.855.

John N. Hooker. A search-infer-and-relax framework for integrating solution methods. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Second International Conference, CPAIOR 2005, Prague, Czech Republic, May 30 - June 1, 2005, Proceedings*, pages 243–257, 2005. doi: 10.1007/11493853_19. URL https://doi.org/10.1007/11493853_19.

Sergio Jiménez, Anders Jonsson, and Héctor Palacios. Temporal planning with required concurrency using classical planning. 2015. URL https://www.aaai.org/ocs/index.php/ICAPS/ICAPS15/paper/view/10623/10405.

Mostepha Redouane Khouadjia, Marc Schoenauer, Vincent Vidal, Johann Dréo, and Pierre Savéant. Multi-objective AI planning: Evaluating DAE-YAHSP on a tunable benchmark. *CoRR*, abs/1212.5276, 2012. URL http://arxiv.org/abs/1212.5276.

Derek Long and Maria Fox. Exploiting a graphplan framework in temporal planning. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy*, pages 52–61, 2003. URL http://www.aaai.org/Library/ICAPS/2003/icaps03-006.php.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - The Planning Domain Definition Language. *URL citeseer. ist. psu. edu/ghallab98pddl. html*, 1998.

Drew M. McDermott. The 1998 ai planning systems competition. *AI Magazine*, 21(2):35, Jun. 2000. doi: 10.1609/aimag.v21i2.1506. URL https://www.aaai.org/ojs/index.php/aimagazine/article/view/1506.

Chiara Piacentini, Maria Fox, and Derek Long. Planning with numeric timed initial fluents. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 4196–4197, 2015. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9859.

Masood Feyzbakhsh Rankooh. Using satisfiability for non-optimal temporal planning. In *The 23th International Conference on Automated Planning and Scheduling Doctoral Consortium*, 2013.

Masood Feyzbakhsh Rankooh and Gholamreza Ghassem-Sani. New encoding methods for sat-based temporal planning. In *Proceedings of the Twenty-Third International Conference*

*on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*, 2013. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS13/paper/view/6015.

Masood Feyzbakhsh Rankooh, Ali Mahjoob, and Gholamreza Ghassem-Sani. Using satisfiability for non-optimal temporal planning. In *Logics in Artificial Intelligence - 13th European Conference, JELIA, 2012, Toulouse, France, September 26-28, 2012. Proceedings*, pages 176–188, 2012. doi: 10.1007/978-3-642-33353-8_14. URL https://doi.org/10.1007/978-3-642-33353-8_14.

Oscar Sapena, Alejandro Torreño, and Eva Onaindia. Parallel heuristic search in forward partial-order planning. *Knowledge Eng. Review*, 31(5):417–428, 2016. doi: 10.1017/S0269888916000230. URL https://doi.org/10.1017/S0269888916000230.

Oscar Sapena, Eliseo Marzal, and Eva Onaindia. Tflap: a temporal forward partial-order planner. temporal track of the international planning competition (ipc). https://ipc2018-temporal.bitbucket.io/planner-abstracts/team2.pdf, 2018.

Emre Savas, Maria Fox, Derek Long, and Daniele Magazzeni. Planning using actions with control parameters. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, pages 1185–1193, 2016. doi: 10.3233/978-1-61499-672-9-1185. URL https://doi.org/10.3233/978-1-61499-672-9-1185.

C. E. Shannon. A mathematical theory of communication. *Bell System Tech. Journal 27*, pages 379–423, 1948.

David E. Smith and Daniel S. Weld. Temporal planning with mutual exclusion reasoning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 326–337, 1999. URL http://ijcai.org/Proceedings/99-1/Papers/048.pdf.

Michal Sroka and Derek Long. A cost-based relaxed planning graph heuristic for enhanced metric sensitivity. In *STAIRS 2014 - Proceedings of the 7th European Starting AI Researcher Symposium, Prague, Czech Republic, August 18-22, 2014*, pages 270–279, 2014. doi: 10.3233/978-1-61499-421-3-270. URL https://doi.org/10.3233/978-1-61499-421-3-270.

Atif Talukdar. Temporal inference in forward search temporal planning dissertation abstract. In *The 26th International Conference on Automated Planning and Scheduling Doctoral Consortium*, pages 73–78, 2016.

Atif Talukdar, Maria Fox, and Derek Long. *Pattern Based Temporal Inference In Forward Search Temporal Planning*, volume 1782. CEUR-WS, 1 2017.

Peter van Beek and Xinguang Chen. Cplan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA.*, pages 585–590, 1999. URL `http://www.aaai.org/Library/AAAI/1999/aaai99-083.php`.

Gérard Verfaillie, Cédric Pralet, and Michel Lemaître. Constraint-based modeling of discrete event dynamic systems. *J. Intelligent Manufacturing*, 21(1):31–47, 2010a. doi: 10.1007/s10845-008-0176-3. URL `https://doi.org/10.1007/s10845-008-0176-3`.

Gérard Verfaillie, Cédric Pralet, and Michel Lemaître. How to model planning and scheduling problems using constraint networks on timelines. *The Knowledge Engineering Review*, 25 (3):319–336, 2010b. doi: 10.1017/S0269888910000172.

Vincent Vidal. Yahsp2, keep it simple, stupid. In *Proceedings of the 7th International Planning Competition (IPC-2011)*, pages 83–90, 2011.

Vincent Vidal. Yahsp3 and yahsp3-mt in the 8th international planning competition. In *Proceedings of the 8th International Planning Competition (IPC-2014)*, pages 64–65, 2014.

Vincent Vidal and Hector Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 570–577, 2004. URL `http://www.aaai.org/Library/AAAI/2004/aaai04-091.php`.

Vincent Vidal and Hector Geffner. Solving simple planning problems with more inference and no search. In *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, pages 682–696, 2005. doi: 10.1007/11564751_50. URL `http://dx.doi.org/10.1007/11564751_50`.

David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Trans. Evolutionary Computation*, 1(1):67–82, 1997. doi: 10.1109/4235.585893. URL `https://doi.org/10.1109/4235.585893`.